PHarr XCPC Templates

PHarr

2024 年 8 月 22 日

目录

第一章 编程技巧和基础算法

1.1 读入优化

```
// 快读
int read() {
    int x = 0, f = 1, ch = getchar();
    while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
    if (ch == '-') f = -1, ch = getchar();
    while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
    return x * f;
}
// 关闭同步
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

1.2 C++ 标准输入输出

```
// 设置输出宽度为 x
cout << setw(x) << val;

// 设置保留小数位数 x ，并四舍五入
cout << fixd<< setprecision(x) << val;

//按进制输出
cout << bitset<10>(i); // 二进制
cout << oct << i; // 八进制
cout << dec << i; // 十进制
cout << hex << i; // 十六进制

```

```
// 设置填充符
cout << setw(10) << 1234; // 默认是空格
cout << setw(10) << setfill('0') << 1234; // 设置填充符
//左侧填充
cout<<setw(10)<<setfill('0')<<setiosflags(ios::left)<<123;
//右侧填充
cout<<setw(10)<<setfill('0')<<setiosflags(ios::right)<<123;

// 单个字符
ch = cin.get();
cout.put(ch);

// 指定长度字符串读入
cout.get(str,80,'a');// 字符串 字符个数 终止字符

// 整行读入
getlin( cin , s );
```

1.3 倍增

天才 ACM

给定一个整数 M，对于任意一个整数集合 S，定义“校验值”如下:

从集合 S 中取出 M 对数 (即 $2\times M$ 个数，不能重复使用集合中的数，如果 S 中的整数不够 M 对，则取到不能取为止)，使得“每对数的差的平方”之和最大，这个最大值就称为集合 S 的“校验值”。

现在给定一个长度为 N 的数列 A 以及一个整数 T。我们要把 A 分成若干段，使得每一段的“校验值”都不超过 T。求最少需要分成几段。

1. 初始化`p = 1 , r = l = 1`
2. 求出`[l,r+p]`这一段的校验值，若校验值小于等于 T 则`r+=p,p*=2`，否则`p/=2`
3. 重复上一步知道 p 的值变为 0 此时的 r 即为所求

```
#include<bits/stdc++.h>
#define int long long
using namespace std;

int read() {
    int x = 0, f = 1, ch = getchar();
```

```
    while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
    if (ch == '-') f = -1, ch = getchar();
    while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
    return x * f;
}

const int N = 5e5+5;
int a[N] , b[N];
int n , m , t , res;
int query( int l , int r ){
    if( r > n ) return 1e19;
    for( int i = l ; i <= r ; i ++ ) b[i] = a[i];
    sort( b + l , b + r + 1);
    int ans = 0;
    for( int i = l , j = r , t = 1 ; t <= m && i < j ; t ++ , i ++ , j
        -- )
        ans += ( b[i] - b[j] ) * ( b[i] - b[j] );
    return ans;
}

void solve(){
    n = read() , m = read() , t = read() , res = 0;
    for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
    for( int l = 1 , r = 1 , p; l <= n ; l = r + 1 ){
        p = 1 , r = l;
        while( p ){
            if( query( l , r + p ) <= t ) r += p , p *= 2;
            else p /= 2;
        }
        res ++;
    }
    cout << res << "\n";
}

int32_t main() {
    for( int T = read(); T ; T -- ) solve();
    return 0;
```

```
}
```

1.4 语法杂项

1.4.1 define 与 typedef

typedef是用来给类型定义别名，是编译器处理的

#define是字面上进行宏定义用的，是在预处理阶段使用的

```
#define STRING char * //宏定义
STRING name , sign;//声明
char * name , sign;//会被替换成这种的结果,只有 name 是指针
// 所以定义类型时候应该回避 #define 而采用typedef
typedef char * STRING;
char * name , * sign;//会被替换成这种结果
```

1.4.2 fill 数组填充

```
// 一维数组
int a[5];
fill( a , a + 5 , 3 );

// 二维数组
int a[5][4];
fill( a[0] , a[0] + 5 * 4 , 6 );

// vector
vector<int> a;
fill( a.begin() , a.end() , 9 );
```

1.4.3 除法取整

```
template <typename T, typename U>
T ceil(T x, U y) {
    return (x > 0 ? (x + y - 1) / y : x / y);
}
template <typename T, typename U>
T floor(T x, U y) {
    return (x > 0 ? x / y : (x - y + 1) / y);
```

```
}
```

1.4.4 map 自定义哈希

```cpp
// 注意需要以下头文件
#include<unordered_map>
#include<chrono>

struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

unordered_map<int, int, custom_hash> mp;
```

1.4.5 文件 IO

```cpp
# include <fstream>

// 流对象
ifstream  fin; // 读入
ofstream fout; // 输出

// 打开
ifstream fin("a.in");
fin.open("a.in");

```

```
// 关闭
fin.close();

// 检测 EOF 到达结尾返回非 0 值
fin.eof();
```

1.4.6 乘法爆 long long

```
ll mul(ll x, ll y, ll m) {
    x %= m, y %= m;
    ll d = ((long double) x * y / m);
    d = x * y - d * m;
    if (d >= m) d -= m;
    if (d < 0) d += m;
    return d;
}
```

1.4.7 枚举子集

```
// 降序遍历 m 的非空子集, s 是 m 的一个非空子集
for (int s = m; s; s = (s - 1) & m) {

}
```

1.5 Lambda 表达式

以下内容绝大部分使用与c++14及更新的标准

Lambda 的组成部分是

```
[capture] (parameters) mutable -> return-type {statement};
```

首先caputre是捕获列表可以从所在代码块中捕获变量。

什么都不写[]就是不进行任何捕获，[=]是值捕获，[&]是引用捕获，值捕获不能修改变量的值，引用捕获可以。特别的，如果值捕获希望在函数内部修改可以使用mutable关键字

同时捕获列表也可以单独针对某一个变量[a]、[&a]分别是值捕获和引用捕获。当然也可以混用[=,&a]对所有变量值捕获，但a除外，a是引用捕获。

然后就是parameters参数列表和statement函数主体，这里与普通的函数没有区别。

-> return-type，函数范围值类型，如果不写可以自动推断，但是如果有多个return且返回类型不同就会CE c++14之后可以用auto来自动的把函数赋值给变量，c++11中则需要自己写

c++17 之后递归可以这样写

```
auto dfs = [e](auto &&self, int x) -> void {
    for (auto y: e[x])
        self(self, y);
};
dfs(dfs, 1);
```

1.6 MT 19937

mt19937是一个很便捷的随机数生成算法，在 c++11中使用非常便捷

```
mt19937 rd(seed) ; // 这样就填入了一个随机数种子
rd(); // 这样就会返回一个随机数
mt19937_64 rd(); // 相同用法，不过返回是一个 64 位整形
```

如果要在一个闭区间 $[a,b]$ 中随机生成一个数

```
mt19937 mt{random_device()()};
uniform_int_distribution rd(a,b);
x = rd(mt); // 这样会返回一个在$[a,b]$范围内的随机数
```

1.7 常用常数

在<math.h>库中有一些常用的参数

```
#if defined _USE_MATH_DEFINES && !defined _MATH_DEFINES_DEFINED
    #define _MATH_DEFINES_DEFINED
    #define M_E        2.71828182845904523536   // e
    #define M_LOG2E    1.44269504088896340736   // log2(e)
    #define M_LOG10E   0.434294481903251827651  // log10(e)
    #define M_LN2      0.693147180559945309417  // ln(2)
    #define M_LN10     2.30258509299404568402   // ln(10)
    #define M_PI       3.14159265358979323846   // pi
    #define M_PI_2     1.57079632679489661923   // pi/2
    #define M_PI_4     0.785398163397448309616  // pi/4
    #define M_1_PI     0.318309886183790671538  // 1/pi
    #define M_2_PI     0.636619772367581343076  // 2/pi
    #define M_2_SQRTPI 1.12837916709551257390   // 2/sqrt(pi)
    #define M_SQRT2    1.41421356237309504880   // sqrt(2)
```

```
    #define M_SQRT1_2  0.707106781186547524401  // 1/sqrt(2)
#endif
```

但是`<math.h>`并没有默认定义`_USE_MATH_DEFINES`，所以用之前需要先定义（在万能头下貌似已经被定义过了），在开头加上`#define _USE_MATH_DEFINES`即可

在 C++ 20 中新增了`<numbers>`库，里面定义了如下常量

```
numbers::e                // e
numbers::log2e            // log2(e)
numbers::log10e           // log10(e)
numbers::pi               // pi
numbers::inv_pi           // 1 / pi
numbers::inv_sqrtpi       // 1 / sqrt(pi)
numbers::ln2              // ln2
numbers::ln10             // ln 10
numbers::sqrt2            // sqrt(2)
numbers::sqrt3            // sqrt(3)
numbers::inv_sqrt3        // 1 / sqrt(3)
numbers::egamma           // 欧拉常数
numbers::phi              // 环境分隔比常数 (1 + sqrt(5)) / 2
```

1.8 有用的库函数

1. `fabs(x)` 用于计算浮点数的绝对值
2. `exp(x)` 计算 e^x
3. `log(x)` 计算 $\ln(x)$
4. `__lg(x)` 计算 $\lfloor \log_2(x) \rfloor$
5. `__gcd(x,y)` 计算 $gcd(x,y)$, 不过在 C++17 引入了`gcd(x,y)`,`lcm(x,y)`
6. `ceil(x)` 返回 $\lceil x \rceil$
7. `floor(x)` $\lfloor x \rfloor$
8. `next_permutation(begin,end)` 将 $[begin, end)$ 变为下一个排列，如果已经是最后一个排列就返回 0
9. `prev_permutation(begin,end)` 将 $[begin, end)$ 变为上一个排列，如果已经是最后一个排列就返回 0
10. `shuffle(begin,end,gen)` 打乱 $[begin, end)$，`gen`是一个随机数生成器（参考 mt19937）

11. `is_sorted(begin,end)` 判断是否升序排序

12. `max(l),min(l)` 对于数组或列表返回最大最小值，例`max({x,y,z})`

13. `exp2(x)` 计算 2^x

14. `log2(x)` 计算 $\log_2(x)$

15. `hypot(x,y)` 计算 $\sqrt{x^2+y^2}$

16. `rotate(iterator begin, iterator middle, iterator end)`作用是把序列中`begin`和`end`连起来，然后再从`middle`处断开，`middle`作为新的`begin`

`__builtin` 家族,这些内容都是 GNU 私货。如果`x`类型是`long long` 请使用`__builtin_xxxll(x)`

1. `__builtin_popcount(x)` 返回 x 在二进制下 1 的个数。

2. `__builtin_parity(x)` 返回 x 在二进制下 1 的个数的奇偶性

3. `__builtin_ffs(x)` 返回 x 在二进制下最后一个 1 是从后往前第几位

4. `__builtin_ctz(x)` 返回 x 在二进制下后导零的个数

5. `__builtin_clz(x)` 返回 x 在二进制下前导零的个数

1.9 交互题

这里说的交互题只是 STDIO 交互题

交互题往往是不会限制运行时间的，一般是通过限制与 oj 的交换次数。对于 IO 交换的题目，要注意的是在每一次输出后都必须要**刷新输出缓冲**后才可以读入。下面介绍几种语言如何刷新缓冲。

1. C `fflush(stdout)`

2. C++ `fflush(stdout)` 或者`cout << flush` 或者用`cout << endl`输出换行也会自动刷新

3. Java `System.out.flush()`

4. Python `stdout.flush()`

1.10 Windows 下的对拍器

```
:again
data.exe > data.in
std.exe < data.in > std.out
test.exe < data.in > test.out
fc std.out test.out
if not errorlevel 1 goto again
```

1.11 Windows 下运行脚本

```
g++ %1.cpp -o %1 -g -std=c++20 -O2 -Wall -Wl,--stack=268435456 -Wall -DLOCAL && echo compile_successfully && %1.exe
```

1.12 Linux 下的对拍器

```
#!/bin/bash
while true; do
    ./data > data.in
    ./std <data.in >std.out
    ./code <data.in >code.out
    if diff std.out code.out; then
        printf "AC\n"
    else
        printf "Wa\n"
        exit 0
    fi
done
```

1.13 MInt

```
template<class T>
constexpr T power(T a, i64 b) {
    T res = 1;
    for (; b; b /= 2, a *= a)
    if (b % 2) res *= a;
```

```
    return res;
}
template<int P>
struct MInt {
    int x;
    static int Mod;
    constexpr MInt() : x(0) {};
    constexpr MInt(int x) : x(norm(x % getMod())) {};
    constexpr static void setMod(int Mod_) {
        Mod = Mod_;
    }
    constexpr static int getMod() {
        if (P > 0) return P;
        else return Mod;
    }
    constexpr int norm(int x) const {
        if (x < 0) x += getMod();
        if (x >= getMod()) x -= getMod();
        return x;
    }
    constexpr int val() const {
        return x;
    }
    explicit constexpr operator int() const {
        // 隐式类型转换把 MInt转换成 int
        return x;
    }

    constexpr MInt operator-() const {
        MInt res;
        res.x = norm(getMod() - x);
        return res;
    }
    constexpr MInt inv() const {
        assert(x != 0);
        return power(*this, getMod() - 2);
    }
    constexpr MInt &operator*=(MInt rhs) &{
```

```
        x = x * rhs.x % getMod();
        return *this;
    }
    constexpr MInt &operator+=(MInt rhs) &{
        x = norm(x + rhs.x);
        return *this;
    }
    constexpr MInt &operator-=(MInt rhs) &{
        x = norm(x - rhs.x);
        return *this;
    }
    constexpr MInt &operator/=(MInt rhs) &{
        return *this *= rhs.inv();
    }
    friend constexpr MInt operator*(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res *= rhs;
        return res;
    }
    friend constexpr MInt operator+(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res += rhs;
        return res;
    }
    friend constexpr MInt operator-(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res -= rhs;
        return res;
    }
    friend constexpr MInt operator/(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res /= rhs;
        return res;
    }

    friend constexpr std::ostream &operator<<(std::ostream &os, const
       MInt &a) {
        return os << a.val();
```

```
    }
    friend constexpr std::istream &operator>>(std::istream &is, MInt &a) {
        int v;
        is >> v;
        a = MInt(v);
        return is;
    }

    friend constexpr bool operator==(MInt lhs, MInt rhs) {
        return lhs.val() == rhs.val();
    }
    friend constexpr bool operator!=(MInt lhs, MInt rhs) {
        return lhs.val() != rhs.val();
    }
};

using Z = MInt<998244353>;
```

如果不太在意时间的话，可以使用下面的版本

```
struct mint {
    int x;

    mint(int x = 0) : x(x) {}

    mint &operator=(int o) { return x = o, *this; }

    mint &operator+=(mint o) { return (x += o.x) >= mod && (x -= mod), *this; }

    mint &operator-=(mint o) { return (x -= o.x) < 0 && (x += mod), *this; }

    mint &operator*=(mint o) { return x = (i64) x * o.x % mod, *this; }

    mint &operator^=(int b) {
        mint w = *this;
        mint ret(1);
```

```
        for (; b; b >>= 1, w *= w) if (b & 1) ret *= w;
        return x = ret.x, *this;
    }

    mint &operator/=(mint o) { return *this *= (o ^= (mod - 2)); }

    friend mint operator+(mint a, mint b) { return a += b; }

    friend mint operator-(mint a, mint b) { return a -= b; }

    friend mint operator*(mint a, mint b) { return a *= b; }

    friend mint operator/(mint a, mint b) { return a /= b; }

    friend mint operator^(mint a, int b) { return a ^= b; }
};
```

1.14 BigInt

```
struct Bigint {
    // representations and structures
    string a; // 存储数字位数 to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1 otherwise
    // constructors
    Bigint() {} // default constructor
    Bigint(string b) {
        if (b[0] == '-') sign = -1, a = b.substr(1);
        else sign = 1, a = b;
        reverse(a.begin(), a.end());
        while (a.back() == '0') a.pop_back();
    }// constructor for string

    Bigint(int c) {
        if (c < 0) sign = -1, a = to_string(-c);
        else sign = 1, a = to_string(c);
        reverse(a.begin(), a.end());
        while (a.back() == '0') a.pop_back();
    }// constructor for int
```

```

    // some helpful methods
    int size() { // 返回数字位数 returns number of digits
        return a.size();
    }

    Bigint inverseSign() { // changes the sign
        sign *= -1;
        return (*this);
    }

    Bigint normalize(int newSign) { // removes leading 0, fixes sign
        for (int i = a.size() - 1; i > 0 && a[i] == '0'; i--)
            a.erase(a.begin() + i);
        sign = (a.size() == 1 && a[0] == '0') ? 1 : newSign;
        return (*this);
    }

    // assignment operator
    void operator=(string b) { // assigns a string to Bigint
        a = b[0] == '-' ? b.substr(1) : b;
        reverse(a.begin(), a.end());
        this->normalize(b[0] == '-' ? -1 : 1);
    }

    void operator=(int b) { //
        string c = to_string(b);
        (*this) = c;
    }

    // conditional operators
    bool operator<(const Bigint &b) const { // less than operator
        if (sign != b.sign) return sign < b.sign;
        if (a.size() != b.a.size())
            return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.
               size();
        for (int i = a.size() - 1; i >= 0; i--)
            if (a[i] != b.a[i])
```

```
                return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
        return false;
    }

    bool operator<(const int &b) const {
        return (*this) < Bigint(b);
    }

    bool operator>(const Bigint &b) const {
        return b < (*this);
    }

    bool operator>(const int &b) const {
        return (*this) > Bigint(b);
    }

    bool operator==(const Bigint &b) const { // operator for equality
        return a == b.a && sign == b.sign;
    }

    bool operator==(const int &b) const {
        return (*this) == Bigint(b);
    }

    bool operator<=(const Bigint &b) const {
        return (*this) < b or (*this) == b;
    }

    bool operator<=(const int &b) const {
        return (*this) <= Bigint(b);
    }

    bool operator>=(const Bigint &b) const {
        return (*this) > b or (*this) == b;
    }

    bool operator>=(const int &b) const {
        return (*this) >= Bigint(b);
```

```
    }

    // mathematical operators
    Bigint operator+(Bigint b) { // addition operator overloading
        if (sign != b.sign) return (*this) - b.inverseSign();
        Bigint c;
        for (int i = 0, carry = 0; i < a.size() || i < b.size() ||
           carry; i++) {
            carry += (i < a.size() ? a[i] - 48 : 0) + (i < b.a.size()
               ? b.a[i] - 48 : 0);
            c.a += (carry % 10 + 48);
            carry /= 10;
        }
        return c.normalize(sign);
    }

    Bigint operator+(int b) {
        return (*this) + Bigint(b);
    }

    Bigint operator-(Bigint b) { // subtraction operator overloading
        if (sign != b.sign) return (*this) + b.inverseSign();
        int s = sign;
        sign = b.sign = 1;
        if ((*this) < b) return ((b - (*this)).inverseSign()).
           normalize(-s);
        Bigint c;
        for (int i = 0, borrow = 0; i < a.size(); i++) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }

    Bigint operator-(int b) {
        return (*this) - Bigint(b);
    }
```

```
    Bigint operator*(Bigint b) { // multiplication operator overloading
        Bigint c("0");
        for (int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48) {
            while (k--) c = c + b; // ith digit is k, so, we add k times
            b.a.insert(b.a.begin(), '0'); // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }

    Bigint operator*(int b) {
        return (*this) * Bigint(b);
    }

    Bigint operator/(Bigint b) { // division operator overloading
        if (b.size() == 1 && b.a[0] == '0') b.a[0] /= (b.a[0] - 48);
        Bigint c("0"), d;
        for (int j = 0; j < a.size(); j++) d.a += "0";
        int dSign = sign * b.sign;
        b.sign = 1;
        for (int i = a.size() - 1; i >= 0; i--) {
            c.a.insert(c.a.begin(), '0');
            c = c + a.substr(i, 1);
            while (!(c < b)) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }

    Bigint operator/(int b) {
        assert(b != 0);
        return (*this) / Bigint(b);
    }

    Bigint operator%(Bigint b) { // modulo operator overloading
        if (b.size() == 1 && b.a[0] == '0') b.a[0] /= (b.a[0] - 48);
```

```
        Bigint c("0");
        b.sign = 1;
        for (int i = a.size() - 1; i >= 0; i--) {
            c.a.insert(c.a.begin(), '0');
            c = c + a.substr(i, 1);
            while (!(c < b)) c = c - b;
        }
        return c.normalize(sign);
    }

    Bigint operator%(int b) {
        assert(b != 0);
        return (*this) % Bigint(b);
    }

    Bigint operator^(Bigint y) {
        Bigint ans("1"), x = (*this);
        while (y > 1) {
            if (y % 2 == 1) ans = ans * x;
            x = x * x, y = y / 2;
        }
        return ans;
    }

    Bigint operator^(int y) {
        Bigint ans("1"), x = (*this);
        while (y > 1) {
            if (y % 2 == 1) ans = ans * x;
            x = x * x, y = y / 2;
        }
        return ans;
    }

    // output method
    void print() {
        if (sign == -1) cout << '-';
        for (int i = a.size() - 1; i >= 0; i--) cout << a[i];
    }
```

```
};

istream &operator>>(istream &is, Bigint &x) {
    string y;
    is >> y;
    x = y;
    return is;
}

ostream &operator<<(ostream &os, const Bigint &x) {
    if(x.a.size() == 0) return os << "0";
    if (x.sign == -1) os << '-';
    for (int i = x.a.size() - 1; i >= 0; i--) os << x.a[i];
    return os;
}
```

第二章　数据结构

2.1　并查集

```
// 初始化
for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
// 查找
int getFa( int x ){
    if( fa[x] == x ) return x;
    return fa[x] = getFa( fa[x] );
}
// 合并
void merge( int x , int y ){
    fa[getFa(x) ] = getFa(y);
}

/*
 * 下面是一种按秩合并的写法
 * 简单来说fa[x]<0表示该点为根结点，当x为根节点时 fa[x] = -size[x]
 */
class dsu{
private:
    vector<int> fa;
public:
    dsu( int n = 1 ){
        fa = vector<int>( n+1 , -1 ) , fa[0] = 0;
    }
    int getfa( int x ){
        if( fa[x] < 0 ) return x;
        return fa[x] = getfa( fa[x] );
    }
    void merge( int x , int y ){
```

```
        x = getfa(x) , y = getfa(y);
        if( x == y ) return ;
        if( fa[x] > fa[y] ) swap( x , y );
        fa[x] += fa[y] , fa[y] = x;
    }
    bool same( int x , int y ){
        x = getfa(x) , y = getfa(y);
        return ( x == y );
    }
};
```

2.2 链式前向星

链式前向星又名邻接表，其实现在我已经几乎不会再手写链式前向星而是采用`vector`来代替

```
vector<int> e[N];// 无边权
vector< pair<int,int> > e[N]; 有边权

e[u].push_back(v);// 加边(u,v)
e[u].push_back( { v, w } ); //加有权边 (u,v,w)
// 无向边 反过来再做一次就好

for( auto v : e[u] ){ // 遍历
}
for( auto [ v , w ] : e[u] ) { // 遍历有权边
}
```

2.3 Hash

2.3.1 Hash 表

对数字的 hash

```
for( int i = 1 ; i <= n ; i ++ ) b[i] = a[i]; // 复制数组
sort( b + 1 , b + 1 + n ) , m = unique( b + 1 , b + 1 + n ) - b;// 排序去重
for( int i = 1 ; i <= n ; i ++ )//hash
    a[i] = lower_bound( b + 1 , b + 1 + m , a[i] ) - b;
```

除此之外，如果更加复杂的 hash 全部使用`unordered_map`容器

2.4 栈

2.4.1 包含 Min 函数的栈

一个支持 $O(1)$ 的 $push(), pop(), top(), getmin()$ 的栈

在维护栈的同时维护一个栈来保存历史上每个时刻都最小值

```
struct MinStack{
    stack<int> a , b;
    void push( int x ){
        a.push(x);
        if( b.size() ) b.push( min( x , b.top() ) );
        else b.push(x);
    }
    void pop(){
        a.pop() , b.pop();
    }
    int top(){
        return a.top();
    }
    int getMin(){
        return b.top();
    }
};
```

2.4.2 单调栈

用来 $O(n)$ 的维护出每一个点左侧第一个比他高的点

```
int h[N] , l[N]; // h[i] 用来记录每一个点的高度 l[i] 记录每一个左侧第一个比 i 高的点的位置
stack<int> stk;

h[0] = INF; //为了便于处理把 0 处理为正无穷
stk.push( h[0] );
for( int i = 1 ; i <= n ; i ++ ){
    while( h[i] >= h[ stk.top() ] ) stk.pop();
    l[i] = stk.top() , stk.push(i);
}
```

2.5 ST 表

ST 表解决的问题是没有修改且查询次数较多（10^6）的区间最值查询

f[i][j]表示 i 向后 2^j 个数的最大值

所以 f[i][j] = max(f[i][j-1] , f[i + (1 << j-1)][j-1]

询问首先计算出数最大的 x 满足 $2^x \leq r-l+1$

这样的话 $[l,r]=[l,l+2^x-1]\cup[r-2^x+1,r]$

```
// LOJ10119
const int N = 1e6+5 , logN = 20;
int a[N] , log_2[N] , f[N][ logN + 5 ];
int32_t main() {
    int n = read() , m = read();
    for( int i = 1 ; i <= n ; i ++ )
        a[i] = read();
    log_2[0] = -1; // 这样初始化可以使得 log_2[1] = 0

    for( int i = 1 ; i <= n ; i ++ ) // O(n) 预处理边界条件 和 log2(i)
        f[i][0] = a[i] , log_2[i] = log_2[i>>1] + 1;

    for( int j = 1 ; j <= logN ; j ++ )
        for( int i = 1; i + (1 << j) - 1 <= n ; i ++ )
            f[i][j] = max( f[i][j-1] , f[ i + ( 1 << j - 1 ) ][j-1] );

    for( int l , r , s ; m ; m -- ){
        l = read() , r = read() , s = log_2[ r - l + 1 ];
        printf("%d\n" , max( f[l][s] , f[ r - ( 1 << s ) + 1 ][s] ));
    }
    return 0;
}
```

2.6 树状数组

2.6.1 单点修改，区间查询

```
struct BinaryIndexedTree{
#define lowbit(x) ( x & -x )
    int n;
    vector<int> b;
```

```

    BinaryIndexedTree( int n ) : n(n) , b(n+1 , 0){};
    BinaryIndexedTree( vector<int> &c ){ // 注意数组下标必须从 1 开始
        n = c.size() , b = c;
        for( int i = 1 , fa = i + lowbit(i) ; i <= n ; i ++ , fa = i +
            lowbit(i) )
            if( fa <= n ) b[fa] += b[i];
    }
    void add( int i , int y ){
        for( ; i <= n ; i += lowbit(i) ) b[i] += y;
        return;
    }

    int calc( int i ){
        int sum = 0;
        for( ; i ; i -= lowbit(i) ) sum += b[i];
        return sum;
    }
};
```

2.6.2 区间修改，单点查询

这里用线段树维护一下差分数组就好

```
// op == 1 [l,r] 加上 val
// op == 2 查询位置 l 的值
int32_t main() {
    n = read() , m = read();
    vector<int> t(n+1);
    for( int i = 1 , x = 0 , lst = 0; i <= n ; i ++ ) x = read() , t[i
        ] = x - lst , lst = x ;
    BinaryIndexedTree B(t);
    for( int op , l , r , val; m ; m -- ){
        op = read();
        if( op == 1 ) l = read() , r = read() , val = read() , B.add(
            l , val ) , B.add( r + 1 , - val );
        else l = read() , printf("%d\n" , B.calc(l) );
    }
    return 0;
```

```
}
```

2.7 分块

```
// https://loj.ac/p/6280
#include <bits/stdc++.h>
using namespace std;
#define int long long

int read() {...}
class decompose {
private:
    struct block {
        int l, r, sum, tag;// tag是单点修改时的懒惰标记
        vector<int> val;

        block(int l, int r) : l(l), r(r) {
            sum = tag = 0;
            val = vector<int>();
        }
    };
    int len;
    vector<block> part;
    vector<int> pos;
public:
    decompose(vector<int> &v) {
        len = v.size();
        int t = sqrt(len);
        pos = vector<int>(len + 1);
        for (int i = 1; i <= t; i++) // 预处理区间信息
            part.emplace_back((i - 1) * t + 1, i * t);
        if (part.back().r < len) // 处理结尾零散部分
            part.emplace_back(part.back().r + 1, len);
        for (int i = 1, j = 0; i <= len; i++) {
            if (i > part[j].r) j++;
            part[j].val.emplace_back(v[i - 1]);
            part[j].sum += v[i - 1], pos[i] = j;
        }
```

```
    }

    int getSum(int l, int r) {
        int sum = 0;
        for (int i = pos[l]; i <= pos[r]; i++) {
            if (part[i].l >= l && part[i].r <= r) sum += part[i].sum;
            else
                for( auto j = max(l, part[i].l) - part[i].l; j <= min(r, part[i].r) - part[i].l ; j++ )
                    sum += part[i].val[j] + part[i].tag;
        }
        return sum;
    }

    void update(int l, int r, int d) {
        for (int i = pos[l]; i <= pos[r]; i++) {
            if (part[i].l >= l && part[i].r <= r){
                part[i].tag += d;
                part[i].sum += part[i].val.size() * d;
            }
            else
                for (int j = max(l, part[i].l) - part[i].l; j <= min(r, part[i].r) - part[i].l; j++)
                    part[i].val[j] += d, part[i].sum += d;
        }
    }
};


int32_t main() {
    int n = read();
    vector<int> a(n);
    for (auto &i: a) i = read();
    decompose p(a);
    for (int opt, l, r, c, sum; n; n--) {
        opt = read(), l = read(), r = read(), c = read();
        if (opt == 0)
            p.update(l, r, c);
```

```
        else {
            c++, sum = p.getSum(l, r), sum = (sum % c + c) % c;
            printf("%lld\n", sum);
        }
    }
    return 0;
}
```

2.8 ODT

```
class ODT{
private:
    struct Node{ // 节点存储结构
        int l , r;
        mutable int val;
        Node( int l , int r = 0 , int val = 0 ) :
                l(l) , r(r) , val(val){};
        bool operator < ( Node b ) const {
            return l < b.l;
        }
        int len() const{
            return r - l + 1;
        }
    };
    int len;
    set<Node> s;
public:
    // 两种构造函数
    ODT( const int & n , const vector<int> & a ){
        len = n;
        int t = 1;
        for( int v : a )
            s.insert( Node( t , t , v ) ) , t ++;
    }
    ODT( const int & n , const int a[] ){
        len = n;
        for( int i = 1 ; i <= n ; i ++ )
            s.insert( Node( i , i , a[i] ) );
```

```
    }

    auto split( int x ){ // 区间分裂
        if( x > len ) return s.end();
        auto it = --s.upper_bound( Node( x ) );
        if( it->l == x ) return it;
        int l = it->l , r = it->r , v = it->val;
        s.erase(it);
        s.insert( Node( l , x-1 , v ) );
        return s.insert( Node( x , r , v ) ).first;
    }
    void assign( int l , int r , int v ){ // 区间赋值 平推操作
        auto itr = split( r + 1 ) , itl = split( l );
        s.erase( itl , itr );
        s.insert( Node( l , r , v ) );
    }
    void add( int l , int r , int x ){ // 区间修改
        auto itr = split( r+1 ) , itl = split( l );
        for( auto it = itl ; it != itr ; it ++ )
            it->val +=x;
    }
    int rank( int l , int r , int x ){ // 求区间第 x 大
        auto itr = split(r+1) , itl = split(l);
        vector< pair<int,int> > v; // first 表示 num , second 表示 cnt
        for( auto it = itl ; it != itr ; it ++ )
            v.push_back({ it->val , it->len() } );
        sort( v.begin() , v.end() );
        for( auto [ num , cnt ] : v ){
            if( cnt < x ) x -= cnt;
            else return num;
        }
    }
    void merge( int l , int r ){ // 区间合并 推平
        auto itr = split(r+1) , itl = split(l);
        vector<Node> cur;
        for( auto it = itl ; it != itr ; it ++ ){
            if( cur.empty() || it->val != cur.back().val )
                cur.push_back( *it );
```

```
            else cur.back().r = it->r;
        }
        s.erase( itl , itr );
        for( auto it : cur )
            s.insert( it );
        return;
    }
    int calP( int l , int r , int x , int y ){ // 求区间 x 次方之和
        auto itr = split(r+1) , itl= split(l);
        int ans = 0;
        for( auto it = itl ; it != itr ; it ++ )
            ans = ( ans + power(it->val,x,y) * it->len()%y ) % y;
        return ans % y;
    }
};
```

2.9 线段树

```
#include <bits/stdc++.h>
using namespace std;

#define int long long

const int N = 5e5+5;
int n , m , a[N];

struct Node{
    int l , r , value , add;
    Node * left , * right;
    Node( int l , int r , int value , int add , Node * left , Node *
       right ) :
            l(l) , r(r) , value(value) , add(add) , left(left) , right
               (right) {};
} * root;

int read() {...}
// 建树
Node * build( int l , int r ){
```

```
    if( l == r ) return new Node( l , r , a[l] , 0 , nullptr , nullptr );
    int mid = ( l + r ) >> 1;
    Node * left = build( l , mid ) , * right = build( mid + 1 , r );
    return new Node( l , r , left->value+right->value,0,left , right);
}
// 标记
void mark( int v , Node * cur ){
    cur -> add += v;
    cur -> value += v * ( cur->r - cur->l + 1 );
}
// 标记下传
void pushdown( Node * cur ){
    if( cur -> add == 0 ) return;
    mark( cur -> add , cur -> left ) , mark( cur -> add , cur -> right );
    cur -> add = 0;
}
// 修改
void modify( int l , int r , int v , Node * cur ){
    if( l > cur -> r || r < cur -> l ) return;
    if( l <= cur -> l && r >= cur -> r ){
        mark( v , cur );
        return;
    }
    pushdown( cur );
    int mid = ( cur -> l + cur -> r ) >> 1;
    if( l <= mid ) modify( l , r , v , cur -> left );
    if( r > mid ) modify( l , r , v , cur -> right );
    cur -> value = cur ->left->value + cur ->right->value;
    return ;
}
// 查询
int query( int l , int r , Node * cur ){
    if( l <= cur->l && r >= cur ->r ) return cur->value;
    pushdown( cur );
    int mid = ( cur->l + cur->r ) >> 1 , res = 0;
    if( l <= mid ) res += query( l , r , cur->left );
```

```
    if( r > mid ) res += query( l , r , cur->right );
    return res;

}

int32_t main(){
    n = read() , m = read();
    for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
    root = build( 1 , n );
    for( int i = 1 , opt , l , r ; i <= m ; i ++ ){
        opt = read() , l = read() , r = read();
        if( opt == 1 ) modify( l , r , read() , root );
        else printf("%lld\n" , query( l , r , root ) );
    }
    return 0;
}
```

区间加，区间乘，区间和，区间平方和

```
struct Node {
    int l, r, value, square, add, mul;
    Node *left, *right;

    Node(int l, int r, int value, int square, Node *left, Node *right)
            : l(l), r(r), value(value), square(square), left(left), right(right) {
        add = 0, mul = 1;
    }
} *root;

Node *build(int l, int r, const vector<int> &v) {
    if (l == r) {
        return new Node(l, r, v[l], v[l] * v[l], nullptr, nullptr);
    }
    int mid = (l + r) / 2;
    Node *left = build(l, mid, v);
    Node *right = build(mid + 1, r, v);
    return new Node(l, r, left->value + right->value, left->square + right->square, left, right);
}
```

```

void markMul(int v, Node *cur) {
    cur->mul *= v;
    cur->add *= v;
    cur->value *= v;
    cur->square *= v * v;
    return;
}

void markAdd(int v, Node *cur) {
    cur->square += (cur->r - cur->l + 1) * v * v + 2 * v * cur->value;
    cur->value += v * (cur->r - cur->l + 1);
    cur->add += v;
    return;
}

void pushdown(Node *cur) {
    if (cur->mul != 1) {
        markMul(cur->mul, cur->left);
        markMul(cur->mul, cur->right);
        cur->mul = 1;
    }
    if (cur->add != 0) {
        markAdd(cur->add, cur->left);
        markAdd(cur->add, cur->right);
        cur->add = 0;
    }
    return;
}

int calcValue(int l, int r, Node *cur) {
    if (l > cur->r or r < cur->l) return 0;
    if (l <= cur->l and cur->r <= r) return cur->value;
    pushdown(cur);
    int mid = (cur->l + cur->r) / 2, sum = 0;
    if (l <= mid) sum += calcValue(l, r, cur->left);
    if (r > mid) sum += calcValue(l, r, cur->right);
    return sum;
```

```
}

int calcSquare(int l, int r, Node *cur) {
    if (l > cur->r or r < cur->l) return 0;
    if (l <= cur->l and cur->r <= r) return cur->square;
    pushdown(cur);
    int mid = (cur->l + cur->r) / 2, sum = 0;
    if (l <= mid) sum += calcSquare(l, r, cur->left);
    if (r > mid) sum += calcSquare(l, r, cur->right);
    return sum;
}

void modifyMul(int l, int r, int v, Node *cur) {
    if (l > cur->r or r < cur->l) return;
    if (l <= cur->l and cur->r <= r) {
        markMul(v, cur);
        return;
    }
    pushdown(cur);
    int mid = (cur->l + cur->r) / 2;
    if (l <= mid) modifyMul(l, r, v, cur->left);
    if (r > mid) modifyMul(l, r, v, cur->right);
    cur->value = cur->left->value + cur->right->value;
    cur->square = cur->left->square + cur->right->square;
    return;
}

void modifyAdd(int l, int r, int v, Node *cur) {
    if (l > cur->r or r < cur->l) return;
    if (l <= cur->l and cur->r <= r) {
        markAdd(v, cur);
        return;
    }
    pushdown(cur);
    int mid = (cur->l + cur->r) / 2;
    if (l <= mid) modifyAdd(l, r, v, cur->left);
    if (r > mid) modifyAdd(l, r, v, cur->right);
    cur->value = cur->left->value + cur->right->value;
```

```
    cur->square = cur->left->square + cur->right->square;
    return;
}
```

2.10 差分

离散化差分

```
// 离散化差分这里我用map实现
map<int,int> b;

void update( int l , int r , int v ){
    b[l] += v , b[r+1] -= v;
}

// 求和
for( auto it = b.begin() ; next(it) != b.end() ; it = next(it) )
    next(it)->second += it->second;
```

2.10.1 二维前缀和、差分

```
// 二维前缀和
b[i][j] = b[i-1][j]+b[i][j-1]-b[i-1][j-1] + a[i][j];

// 求 (x1,y1) ~ (x2,y2) 和
b[x2][y2] - b[x2][y1-1] - b[x1-1][y2] + b[x1-1][y1-1];
```

2.11 扫描线

2.11.1 求面积并

求 N 个矩形面积的并，每个矩形用 $(x_a, y_a), (x_b, y_b)$ 表示。

将整个图形分为 $2N$ 部分, 这样的话每个矩形可以用两条线段 $(x_a, y_a, y_b, 1), (x_b, y_a, y_b, -1)$ 表示。

在需要用到扫描线的题目中 y 值通常很大, 甚至可能不是整数, 所以我们需要进行离散化。记 $val(y)$ 为 y 离散化之后的值, $raw(i)$ 为 i 的原始坐标。在离散化之后有 tot 个 y 的坐标值, 分别对应为 $raw(1), raw(2), raw(3), \ldots, raw(tot)$, 则扫描线被分为 $tot-1$ 段, 其中第 i 段为 $[raw(i), raw(i+1)]$。

将线段按照 x 值排序，初始每一段都是 0。然后遍历每个线段 (x_i, y_a, y_b, v), 如果到 x_{i-1} 的线段覆盖的中长度为 len, 则当前矩形的面积为 $(x_i - x_{i-1}) \times len$。然后给 $[val(y_a), val(y_b) - 1]$ 加 v, 相当于覆盖了 $[x_i, x_{i+1}]$ 的部分。

```
// luogu P5490
#include<bits/stdc++.h>
using namespace std;
using i32 = int32_t;
using i64 = long long;
#define int i64
using seg = array<int, 4>; // {x, y1, y2, v}
using vi = vector<int>;

struct Node {
    int l, r, value, cnt;// value 被覆盖子区间长度，cnt 当前区间被完整覆盖的次数。
    Node *left, *right;

    Node(int l, int r, int value, int cnt, Node *left, Node *right)
            : l(l), r(r), value(value), cnt(cnt), left(left), right(right) {};
} *root;

vi raw;// 原始坐标

int val(int x) { // // x 是原始值，返回哈希值
    int i = lower_bound(raw.begin(), raw.end(), x) - raw.begin();
    if (raw[i] != x) return -1;
    return i;
}

Node *build(int l, int r) {
    if (l == r) return new Node(l, r, 0, 0, nullptr, nullptr);
    int mid = (l + r) / 2;
    auto left = build(l, mid), right = build(mid + 1, r);
    return new Node(l, r, 0, 0, left, right);
}

void maintain(Node *cur) {
```

```
    if (cur->cnt > 0) cur->value = raw[cur->r + 1] - raw[cur->l];
    else if (cur->left == nullptr) cur->value = 0;
    else cur->value = cur->left->value + cur->right->value;
    return;
}

void modify(int l, int r, int v, Node *cur) {
    if (l > cur->r or r < cur->l) return;
    if (l <= cur->l and cur->r <= r) {
        cur->cnt += v;
        maintain(cur);
        return;
    }
    int mid = (cur->l + cur->r) / 2;
    if (l <= mid) modify(l, r, v, cur->left);
    if (r > mid) modify(l, r, v, cur->right);
    maintain(cur);
    return;
}

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vector<seg> p;
    for (int i = 1, xa, xb, ya, yb; i <= n; i++) {
        cin >> xa >> ya >> xb >> yb;
        p.push_back(seg{xa, ya, yb, 1});
        p.push_back(seg{xb, ya, yb, -1});
        raw.push_back(ya), raw.push_back(yb);
    }
    sort(p.begin(), p.end());
    sort(raw.begin(), raw.end());
    raw.resize(unique(raw.begin(), raw.end()) - raw.begin());
    int tot = raw.size() - 1;
    root = build(0, tot);
    modify(val(p[0][1]), val(p[0][2]) - 1, p[0][3], root);
    int res = 0;
```

```
    for (int i = 1; i < p.size(); i++) {
        res += (p[i][0] - p[i - 1][0]) * root->value;
        modify(val(p[i][1]), val(p[i][2]) - 1, p[i][3], root);
    }
    cout << res;
    return 0;
}
```

2.11.2 二维数点

单纯的二维数点数点问题，可以只用树状数组就可以维护。

$d(x,y)$ 表示从 $(0,0)$ 到 (x,y) 中点的数量，因此从左下角 (a,b) 到右上角 (c,d) 中点的数量就可以表示为 $d(c,d)-d(c,b-1)-d(a-1,d)+d(a-1,b-1)$，这个形式就是普通的二维前缀和。我们把式子稍作变形转换为 $d((c,d)-d(c,b-1))-(d(a-1,d)-d(a-1,b-1))$ 这样的话就可以用扫描线优化掉一维。

```
#include<bits/stdc++.h>

using namespace std;

using vi = vector<int>;
using pii = pair<int, int>;
using seg = array<int, 5>;

#define lowbit(x) ( x & -x )

const int N = 1e7 + 5;

struct BinaryIndexedTree {...}; // 树状数组板子


int32_t main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<pii> a(n);
    for (auto &[x, y]: a)
        cin >> x >> y, x++, y++;
    vector<seg> p;
```

```
    for (int i = 0, xa, xb, ya, yb; i < m; i++) {
        cin >> xa >> ya >> xb >> yb;
        xa++, ya++, xb++, yb++;
        p.push_back(seg{xb, ya, yb, i, 1});
        p.push_back(seg{xa - 1, ya, yb, i, -1});
    }

    sort(p.begin(), p.end());
    sort(a.begin(), a.end());
    vi res(m);
    BinaryIndexedTree bit(N);
    for (int j = 0; const auto &it: p) {
        while (j < a.size() and a[j].first <= it[0])
            bit.update(a[j].second, 1), j++;
        res[it[3]] += it[4] * bit.calc(it[1], it[2]);
    }

    for (auto i: res)
        cout << i << "\n";
    return 0;
}
```

2.12 Splay

2.12.1 普通平衡树

您需要写一种数据结构，来维护一些数，其中需要提供以下操作：

1. 插入一个数 x。
2. 删除一个数 x（若有多个相同的数，应只删除一个）。
3. 定义**排名**为比当前数小的数的个数 $+1$。查询 x 的排名。
4. 查询数据结构中排名为 x 的数。
5. 求 x 的前驱（前驱定义为小于 x，且最大的数）。
6. 求 x 的后继（后继定义为大于 x，且最小的数）。

对于操作 3,5,6，**不保证**当前数据结构中存在数 x。

```cpp
// luogu P3369
#include<bits/stdc++.h>

using namespace std;

const int N = 1e5 + 5;

struct Splay {
    int root, tot;

    array<int, N> f, val, cnt, size;
    array <array<int, 2>, N> c;

    void updateSize(int p) {
        size[p] = size[c[p][0]] + size[c[p][1]] + cnt[p];
        return;
    }

    bool get(int p) { // 判断 p 是不是父亲的右儿子
        return p == c[f[p]][1];
    }

    void clear(int p) { // 清空节点
        c[p][0] = c[p][1] = f[p] = val[p] = size[p] = cnt[p] = 0;
    }

    int newNode(int k) {
        ++tot;
        val[tot] = k, cnt[tot] = 1;
        return tot;
    }

    void rotate(int p) {
        int fp = f[p], ffp = f[fp], chk = get(p);
        c[fp][chk] = c[p][chk ^ 1];
        if (c[p][chk ^ 1]) f[c[p][chk ^ 1]] = fp;
        c[p][chk ^ 1] = fp;
        f[fp] = p, f[p] = ffp;
```

```
        if (ffp) c[ffp][fp == c[ffp][1]] = p;
        updateSize(fp), updateSize(p);
    }

//    如果只换到根节点可以简写
//    int splay(int p) {
//        for (int fp = f[p]; fp = f[p], fp; rotate(p))
//            if (f[fp]) rotate(get(p) == get(fp) ? fp : p);
//        return root = p;
//    }

    void splay(int p, int goal = 0) {
        if (goal == 0) root = p;
        while (f[p] != goal) {
            int fp = f[p], ffp = f[fp];
            if (ffp != goal) {
                if (get(fp) == get(p))
                    rotate(fp);
                else
                    rotate(p);
            }
            rotate(p);
        }
    }

    void insert(int k) {
        if (root == 0) {
            root = newNode(k);
            updateSize(root);
            return;
        }
        int p = root, fp = 0;
        while (true) {
            if (val[p] == k) {
                cnt[p]++;
                updateSize(p), updateSize(fp);
                splay(p);
                break;
```

```
            }
            fp = p, p = c[p][val[p] < k];
            if (p == 0) {
                p = newNode(k);
                f[p] = fp, c[fp][val[fp] < k] = p;
                updateSize(p), updateSize(fp);
                splay(p);
                break;
            }
        }
    }

    int rank(int k) {
        int ans = 0, p = root;
        while (true) {
            if (k < val[p]) {
                p = c[p][0];
            } else {
                ans += size[c[p][0]];
                if (p == 0) return ans + 1;
                if (k == val[p]) {
                    splay(p);
                    return ans + 1;
                }
                ans += cnt[p];
                p = c[p][1];
            }
        }
    }

    int kth(int k) {
        int p = root;
        while (true) {
            if (c[p][0] and k <= size[c[p][0]]) {
                p = c[p][0];
            } else {
                k -= cnt[p] + size[c[p][0]];
                if (k <= 0) {
```

```
                    splay(p);
                    return val[p];
                }
                p = c[p][1];
            }
        }
    }

    int pre() {
        int p = c[root][0];
        if (p == 0) return p;
        while (c[p][1]) p = c[p][1];
        splay(p);
        return p;
    }

    int suf() {
        int p = c[root][1];
        if (p == 0) return p;
        while (c[p][0]) p = c[p][0];
        splay(p);
        return p;
    }

    void del(int k) {
        rank(k);
        if (cnt[root] > 1) {
            cnt[root]--;
            updateSize(root);
            return;
        }
        if (c[root][0] == 0 and c[root][1] == 0) {
            clear(root);
            root = 0;
            return;
        }
        if (c[root][0] == 0) {
            int p = root;
```

```
            root = c[root][1];
            f[root] = 0;
            clear(p);
            return;
        }
        if (c[root][1] == 0) {
            int p = root;
            root = c[root][0];
            f[root] = 0;
            clear(p);
            return;
        }
        int p = root;
        int x = pre();
        f[c[p][1]] = x;
        c[x][1] = c[p][1];
        clear(p);
        updateSize(root);
    }

    int pre(int k) {
        insert(k);
        int ans = val[pre()];
        del(k);
        return ans;
    }

    int suf(int k) {
        insert(k);
        int ans = val[suf()];
        del(k);
        return ans;
    }
};

int main() {
    int n;
    cin >> n;
```

```
    Splay tree;
    for (int opt, x; n; n--) {
        cin >> opt >> x;
        if (opt == 1) {
            tree.insert(x);
        } else if (opt == 2) {
            tree.del(x);
        } else if (opt == 3) {
            cout << tree.rank(x) << "\n";
        } else if (opt == 4) {
            cout << tree.kth(x) << "\n";
        } else if (opt == 5) {
            cout << tree.pre(x) << "\n";
        } else {
            cout << tree.suf(x) << "\n";
        }
    }
    return 0;
}
```

2.12.2 文艺平衡树

您需要写一种数据结构（可参考题目标题），来维护一个有序数列。

其中需要提供以下操作：翻转一个区间，例如原有序序列是 5 4 3 2 1，翻转区间是 [2,4] 的话，结果是 5 2 3 4 1。

```
#include<bits/stdc++.h>

using namespace std;

const int N = 1e5 + 5;

const int inf = 1e9;

struct Splay {
    int root, tot, n;

    array<int, N> f, val, cnt, size, reverseTag;
    array<array<int, 2>, N> c;
```

```

    void updateSize(int p) {
        size[p] = size[c[p][0]] + size[c[p][1]] + cnt[p];
        return;
    }

    bool get(int p) { // 判断 p 是不是父亲的右儿子
        return p == c[f[p]][1];
    }

    void clear(int p) { // 清空节点
        c[p][0] = c[p][1] = f[p] = val[p] = size[p] = cnt[p] =
           reverseTag[p] = 0;
    }

    int newNode(int k) {
        ++tot;
        val[tot] = k, cnt[tot] = 1;
        return tot;
    }

    void rotate(int p) {
        int fp = f[p], ffp = f[fp], chk = get(p);
        c[fp][chk] = c[p][chk ^ 1];
        if (c[p][chk ^ 1]) f[c[p][chk ^ 1]] = fp;
        c[p][chk ^ 1] = fp;
        f[fp] = p, f[p] = ffp;
        if (ffp) c[ffp][fp == c[ffp][1]] = p;
        updateSize(fp), updateSize(p);
    }

    void splay(int p, int goal = 0) {
        if (goal == 0) root = p;
        while (f[p] != goal) {
            int fp = f[p], ffp = f[fp];
            if (ffp != goal) {
                if (get(fp) == get(p))
                    rotate(fp);
```

```
                else
                    rotate(p);
            }
            rotate(p);
        }
    }

    void insert(int k) {
        if (root == 0) {
            root = newNode(k);
            updateSize(root);
            return;
        }
        int p = root, fp = 0;
        while (true) {
            if (val[p] == k) {
                cnt[p]++;
                updateSize(p), updateSize(fp);
                splay(p);
                break;
            }
            fp = p, p = c[p][val[p] < k];
            if (p == 0) {
                p = newNode(k);
                f[p] = fp, c[fp][val[fp] < k] = p;
                updateSize(p), updateSize(fp);
                splay(p);
                break;
            }
        }
    }

    int rank(int k) {
        int ans = 0, p = root;
        while (true) {
            if (k < val[p]) {
                p = c[p][0];
            } else {
```

```
                ans += size[c[p][0]];
                if (p == 0) return ans + 1;
                if (k == val[p]) {
                    splay(p);
                    return ans + 1;
                }
                ans += cnt[p];
                p = c[p][1];
            }
        }
    }

    int kth(int k) {
        int p = root;
        while (true) {
            pushdown(p);
            if (c[p][0] and k <= size[c[p][0]]) {
                p = c[p][0];
            } else {
                k -= cnt[p] + size[c[p][0]];
                if (k <= 0) {
                    splay(p);
                    return p;
                }
                p = c[p][1];
            }
        }
    }

    int pre() {
        int p = c[root][0];
        if (p == 0) return p;
        while (c[p][1]) p = c[p][1];
        splay(p);
        return p;
    }

    int suf() {
```

```
        int p = c[root][1];
        if (p == 0) return p;
        while (c[p][0]) p = c[p][0];
        splay(p);
        return p;
    }

    void del(int k) {
        rank(k);
        if (cnt[root] > 1) {
            cnt[root]--;
            updateSize(root);
            return;
        }
        if (c[root][0] == 0 and c[root][1] == 0) {
            clear(root);
            root = 0;
            return;
        }
        if (c[root][0] == 0) {
            int p = root;
            root = c[root][1];
            f[root] = 0;
            clear(p);
            return;
        }
        if (c[root][1] == 0) {
            int p = root;
            root = c[root][0];
            f[root] = 0;
            clear(p);
            return;
        }
        int p = root;
        int x = pre();
        f[c[p][1]] = x;
        c[x][1] = c[p][1];
        clear(p);
```

```
        updateSize(root);
    }

    int pre(int k) {
        insert(k);
        int ans = val[pre()];
        del(k);
        return ans;
    }

    int suf(int k) {
        insert(k);
        int ans = val[suf()];
        del(k);
        return ans;
    }

    void tagReverse(int p) {
        swap(c[p][0], c[p][1]);
        reverseTag[p] ^= 1;
    }

    void pushdown(int p) {
        if (reverseTag[p]) {
            tagReverse(c[p][0]);
            tagReverse(c[p][1]);
            reverseTag[p] = 0;
        }
    }

    void reverse(int l, int r) {
        int L = kth(l), R = kth(r + 2);
        splay(L), splay(R, L);
        int tmp = c[c[L][1]][0];
        tagReverse(tmp);
    }

    int build(int l, int r, int fp, const vector<int> &a) {
```

```
        if (l > r) return 0;
        int mid = (l + r) / 2;
        int p = newNode(a[mid]);
        f[p] = fp;
        c[p][0] = build(l, mid - 1, p, a);
        c[p][1] = build(mid + 1, r, p, a);
        updateSize(p);
        return p;
    }

    void init(vector<int> a) { // a 数组 1 base
        n = a.size() - 1;
        a.push_back(inf);
        root = build(0, n + 1, 0, a);
        return;
    }

    void display(int p) {
        pushdown(p);
        if (c[p][0]) display(c[p][0]);
        if (val[p] >= 1 and val[p] <= n) cout << val[p] << " ";
        if (c[p][1]) display(c[p][1]);
    }

    void display() {
        display(root);
        cout << "\n";
    }
};

int main() {
    int n, m;
    cin >> n >> m;
    Splay tree;
    vector<int> a(n + 1);
    iota(a.begin(), a.end(), 0);
    tree.init(a);
    for (int l, r; m; m--) {
```

```
        cin >> l >> r;
        tree.reverse(l, r);
    }
    tree.display();
    return 0;
}
```

第三章　图论

3.1　拓扑排序

在一个 DAG 中，将图中的顶点以线性的方式排序，使得对于任何一条 u 到 v 的有向边，u 都可以出现在 v 的前面

拓扑序判环如果图中已经没有入度为零的点但是依旧有点时，即说明图中存在环。

拓扑序判链如果求拓扑序的过程中队列中同时存在两个及以上的元素，说明拓扑序不唯一，不是一条链

字典序最大、最小的拓扑序把 Kahn 中队列换成是大根堆、小根堆实现的优先队列就好

3.1.1　DFS 算法

```
vector<int> e[N]; // 邻接表
vector<int> topo; // 存储拓扑序
set<int> notInDeg;// 储存没有入读的点
int vis[N];

bool dfs( int u ){
    vis[u] = 1;
    for( auto v : e[u] ){
        if( vis[v] )  return 0;
        if( !dfs(v) ) return 0;
    }
    topo.push_back(u);
    return 1;
}

bool topSort(){
    if( notInDeg.empty() ) return 0 ;
    for( int u : notInDeg ){
        if( !dfs(u) ) return 0;
    }
```

```
    reverse( topo.begin() , topo.end() );
    return 1;
}
```

3.1.2 Kahn 算法

```
vector<int> e[N]; // 邻接表
vector<int> topo; // 存储拓扑序
int inDeg[N]; // 记录点的当前入度

bool topSort(){
    queue<int> q;
    for( int i = 1 ; i <= n ; i ++ )
        if( inDeg[i] == 0 ) q.push(i);
    while( q.size() ){
        int u = q.front() ; q.pop();
        topo.push_back(u);
        for( auto v : e[u] ){
            if( --inDeg[v] == 0 ) q.push(v);
        }
    }
    return topo.size() == n;
}
```

3.2 最近公共祖先 (LCA)

3.2.1 向上标记法

向上标价法最差的复杂度是 $O(N)$ 的，适用于求 LCA 次数少的情况，代码非常好写

```
\\ luogu P3379
const int N = 5e5+5;
int n , m , sta , dep[N] , fa[N];
vector<int> e[N];

int read() {...}
void dfs( int x ){
    for( auto v : e[x] ){
        if( v == fa[x] ) continue;
```

```
        dep[v] = dep[x] + 1 , fa[v] =x;
        dfs(v);
    }
}

int lca( int x , int y ){
    if( dep[x] < dep[y] ) swap( x , y );
    while( dep[x] > dep[y] ) x = fa[x];
    while( x != y ) x = fa[x] , y = fa[y];
    return x;
}

int32_t main() {
    n = read() - 1 , m = read() , sta = read();
    for( int u , v ; n ; n -- )
        u = read(),v = read() , e[u].push_back(v) , e[v].push_back(u);
    dep[sta] = 0 , fa[sta] = sta;
    dfs( sta );
    for( int x , y ; m ; m -- ){
        x = read() , y = read();
        cout << lca(x,y) << endl;
    }
    return 0;
}
```

3.2.2 倍增 LCA

值得注意的是向上倍增的过程中`for( int i = t ; i >= 0 ; i --)`不能写错

```
const int N = 5e5+5;
int n , m , sta , logN , dep[N] , fa[N][20];
vector<int> e[N];
int read() {...}

void dfs( int x ){
    for( auto v : e[x] ){
        if( dep[v] ) continue;
        dep[v] = dep[x] + 1 , fa[v][0] = x;
        for( int i = 1 ; i <= logN ; i ++ )
```

```
                fa[v][i] = fa[ fa[v][i-1] ][i-1];
            dfs(v);
    }
}
int lca( int x , int y ){
    if( dep[x] > dep[y] ) swap( x , y );
    for( int i = logN ; i >= 0 ; i -- )
        if( dep[ fa[y][i] ] >= dep[x] ) y = fa[y][i];
    if( x == y ) return x;
    for( int i = logN ; i >= 0 ; i -- ){
        if( fa[x][i] != fa[y][i] ) x = fa[x][i] , y = fa[y][i];
    }
    return fa[x][0];
}

int32_t main() {
    n=read()-1 , m=read() , sta=read() , logN =(int)log2(n) + 1;
    int k = n;
    for( int u , v ; n ; n -- )
        u=read() , v=read() , e[u].push_back(v) , e[v].push_back(u);
    dep[sta] = 1;
    dfs( sta );
    for( int x , y ; m ; m -- ){
        x = read() , y = read();
        cout << lca(x,y) << endl;
    }
    return 0;
}
```

3.2.3 RMQ 求 LCA

dfs 求出深度和 dfn。位置在 $[dfn_{u+1}, dfn_v]$ 之间深度最小的节点的父亲就是 LCA。区间最值用 ST 表维护。

```
// luogu P3379
#include <bits/stdc++.h>
using namespace std;

int dn;
```

```
vector<int> dfn;
vector <vector<int>> e, f;

void dfs(int x, int fa) {
    f[0][dfn[x] = ++dn] = fa;
    for (auto y: e[x])
        if (y != fa) dfs(y, x);
}
int get(int x, int y) {
    if (dfn[x] < dfn[y]) return x;
    return y;
}
int lca(int u, int v) {
    if (u == v) return u;
    u = dfn[u], v = dfn[v];
    if (u > v) swap(u, v);
    int d = log2(v - u++);
    return get(f[d][u], f[d][v - (1 << d) + 1]);
}
int32_t main() {
    ios::sync_with_stdio(0), cin.tie(0);
    int n, m, root, logN;
    cin >> n >> m >> root;
    e.resize(n + 1), dfn.resize(n + 1);
    logN = log2(n);
    f = vector(logN + 1, vector<int>(n + 1));
    for (int i = 1, x, y; i < n; i++)
        cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
    dfs(root, 0);

    for (int i = 1; i <= logN; i++)
        for (int j = 1; j + (1 << i) - 1 <= n; j++)
            f[i][j] = get(f[i - 1][j], f[i - 1][j + (1 << i - 1)]);

    for (int x, y; m; m--) cin >> x >> y, cout << lca(x, y) << "\n";
    return 0;
}
```

3.3 最短路

3.3.1 Floyd

```
int dis[N][N];
for( int i = 1 ; i <= n ; i ++ )
    for( int j = 1 ; j < i ; j ++ )
        dis[i][j] = dis[j][i] = inf;

for( int u , v , w ; m ; m -- )
    u = read() , v = read() , w = read() , dis[u][v] = dis[v][u] = w;
for( int k = 1 ; k <= n ; k ++ )
    for( int i = 1 ; i <= n ; i ++ )
        for( int j = 1 ; j < i ; j ++ )
            f[i][j] = f[j][i] = min( f[i][j] , f[i][k] + f[k][j] );
```

3.3.2 Dijkstra

复杂度是 $O(m\log(n))$

```
void dij(){
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    dis[sta] = 0;
    priority_queue< pair<int,int> , vector<pair<int,int>> , greater<
       pair<int,int>> > q;
    q.push( { 0 , sta } );
    while( q.size() ){
        int u = q.top().second ; q.pop();
        if( vis[u] ) continue;
        vis[u] = 1;
        for( auto [v,w] : e[u] )
            if( dis[v] > dis[u] + w ){
                dis[v] = dis[u] + w;
                q.push( {dis[v] , v } );
            }
    }
}
```

3.3.3 Bellman-Ford

复杂度是 $O(km)$

```
bool bellmanFord(){ // 返回是否有最短路
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    dis[sta] = 0;
    bool flag;
    for( int i = 1 ; i <= n ; i ++ ){
        flag = 0;
        for( int u = 1 ; u <= n ; u ++ ){
            if( dis[u] == inf ) continue; // 如果当前点和起点没有联通，就无法进行松弛操作
            for( auto [v,w] : e[u] ){
                if( dis[v] <= dis[u] + w ) continue;
                dis[v] = dis[u] + w , flag = 1;// 记录时候进行松弛操作
            }
        }
        if( !flag ) break;
    }
    return flag;
}
```

3.3.4 SPFA

没有准确复杂度，下限是 $O(m\log(n))$，上限是 $O(nm)$

```
int dis[N] , cnt[N];
vector< pair<int,int> > e[N];
bitset<N> vis;

bool spfa(){
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    queue< int > q;
    dis[sta] = 0 , vis[sta] = 1 , q.push(sta);
    for( int u ; q.size() ; ){
        u = q.front() , q.pop() , vis[u] = 0;
        for( auto [ v , w ] : e[u] ){
            if( dis[v] <= dis[u] + w ) continue;
            dis[v] = dis[u] + w ;
            cnt[v] = cnt[u] + 1; // 记录最短路经过了几条边
```

```
            if( cnt[v] >= n ) // 最短路最长是 n-1
                return false; // 此时说明出现了 负环
            if( !vis[v] ) vis[v] = 1 , q.push(v);
        }
    }
    return true;
}
```

3.4 最小生成树

3.4.1 Kruskal

Kruskal 总是维护无向图的最小生成深林。

```
// Luogu P3366
const int N = 5005;
int n , m , fa[N] , cnt = 0 , sum;

int read() {...}

int getfa( int x ){...}

void merge( int x , int y ){...} // 并查集合并

int32_t main(){
    n = read() , m = read();
    for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
    vector<tuple<int,int,int>> e(m);
    for( auto & [ w , u , v ] : e )
        u = read() , v = read() , w = read();
    sort( e.begin() , e.end() );
    for( auto[ w , u , v ] : e ){
        if( getfa(u) == getfa(v) ) continue;
        merge( u , v ) , sum += w , cnt ++;
        if( cnt == n - 1 ) break;
    }
    if( cnt == n - 1 ) cout << sum << "\n";
    else cout << "orz\n";// 图不联通
    return 0;
```

```
}
```

最大生成树只需要把边从大到小选择就好

3.5 树

3.5.1 树的深度

```
dep[sta] = 1;

void dfs( int u ){
    for( auto v : e[u] ){
        if( dep[v] ) continue;
        dep[v] = dep[u] + 1;
        dfs( v );
    }
}

dfs(sta);
```

3.5.2 二叉树还原

```
struct Node {
    int v;
    Node *l, *r;
    Node(int v, Node *l, Node *r) : v(v), l(l), r(r) {};
};
// 根 据 中 序、 后 序 还 原 二 叉 树
Node *build(vector<int> mid, vector<int> suf) {
    int v = suf.back();
    Node *l = nullptr, *r = nullptr;
    int t;
    for (t = 0; t < mid.size(); t++)
        if (mid[t] == v) break;
    auto midll = mid.begin();
    auto midlr = mid.begin() + t;
    auto midrl = mid.begin() + t + 1;
    auto midrr = mid.end();
    auto sufll = suf.begin();
```

```
    auto suflr = suf.begin() + t;
    auto sufrl = suf.begin() + t;
    auto sufrr = suf.end() - 1;

    auto midl = vector<int>(midll, midlr);
    auto midr = vector<int>(midrl, midrr);
    auto sufl = vector<int>(sufll, suflr);
    auto sufr = vector<int>(sufrl, sufrr);

    if (!midl.empty()) l = build(midl, sufl);
    if (!midr.empty()) r = build(midr, sufr);

    return new Node(v, l, r);
}
```

3.5.3 树的 DFS 序和欧拉序

DFS 序中一个点会出现一次，欧拉序会出现两次，两者都是 dfs 时经过的点的顺序，欧拉序中的第二次出现就是点结束搜索是回溯时的顺序

```
// 求DFS序
vector<int> dfsSort;
vector<int> e[N];
bitset<N> vis;

void dfs( int x ){
    dfsSort.push_back(x) , vis[x] = 1;
    for( auto it : e[x] ){
        if( vis[it] ) continue;
        dfs( it );
    }
}
// 求 欧 拉 序
vector<int> eulerSort;
vector<int> e[N];
bitset<N> vis;

void dfs( int x ){
    eulerSort.push_back(x) , vis[x] = 1;
```

```
    for( auto it : e[x] ){
        if( vis[it] ) continue;
        dfs( it );
    }
    eulerSort.push_back(x);
}
```

3.5.4 树的重心

对于size[i]表示每个点子树大小，max_part(x)表示删去x后最大的子树的大小，max_part取到最小值的点p就是树的重心

```
void dfs( int u ){
    vis[u] = size[u] = 1;
    for( auto v : e[u] ){
        if( vis[v] ) continue;
        dfs(v);
        size[x] += size[v];
        max_part = max( max_part , size[v] );
    }
    max_part = max( max_part , n - size[u] );
    if( max_part > ans )
        ans = max_part , pos = u;
}
```

3.5.5 树上前缀和

设sum[i]表示节点i到根节点的权值总和。

如果是点权，x,y路径上的和为sum[x]+sum[y]-sum[lca]-sum[fa[lca]]

```
// Loj 2491
// 一颗树根节点是 1 , 点权就是深度的 k 次方
// m次询问,每次问(u,v)路径上点权之和
// k 每次都不同但是取值范围只有[1,50]
#define int long long
const int N = 3e5+5 , mod = 998244353;
int n , sum[N][55] , fa[N][20] , dep[N] , logN;
vector<int> e[N];

int read(){...}
```

```

void dfs( int x ){
    for( auto v : e[x] ){
        if( v == fa[x][0] ) continue;
        dep[v] = dep[x] + 1 , fa[v][0] = x;
        for( int i = 1 , val = 1; i <= 50 ; i ++ )
            val = val * dep[v]% mod , sum[v][i] = (val + sum[x][i]) %
                mod;
        for( int i = 1 ; i <= logN ; i ++ )
            fa[v][i] = fa[ fa[v][i-1] ][i-1];
        dfs(v);
    }
}

int lca( int x , int y ){...}

int32_t main(){
    n = read() , logN = (int)log2(n)+1;
    for( int i = 2 , u , v ; i <= n ; i ++ )
        u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
            );
    dep[1] = 0;
    dfs( 1 );
    for( int m = read() , u , v , k , t ; m ; m -- ){
        u = read() , v = read() , k = read() , t = lca( u , v );
        cout << (sum[u][k]+sum[v][k]-sum[t][k]-sum[fa[t][0]][k]+2*mod)
            %mod << "\n";
    }
    return 0;
}
```

如果是边权，x,y路径上的和为sum[x]+sum[y]-2*sum[lca]

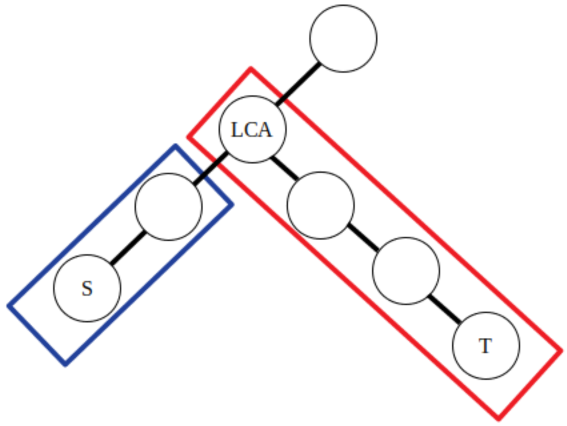
```
//LOJ 10134 树上前缀和 边
const int N = 1e4+5;

int n , m , sum[N] , logN , dep[N] , fa[N][15];
vector<pair<int,int>> e[N];

int read(){...}
```

```

void dfs( int x ){ // 这里和 lca 极其类似，就是在多维护了一个前缀和
    for( auto [v,w] : e[x] ){
        if( dep[v] ) continue;
        dep[v] = dep[x] + 1 , fa[v][0] = x , sum[v] = sum[x] + w;
        for( int i = 1 ; i <= logN ; i ++ )
            fa[v][i] = fa[ fa[v][i-1] ][ i-1 ];
        dfs(v);
    }
}

int lca( int x , int y ){...}//这里就是 lca 的板子

int main(){
    n = read() , m = read() ,logN = (int)log2(n)+1;
    for( int i = 2 , u , v , w ; i <= n ; i ++ )
        u = read() , v = read() , w = read() , e[u].push_back( {v,w} )
            , e[v].push_back( {u,w} );
    dep[1] = 1 , dfs(1);
    for( int u , v ; m ; m -- ){
        u = read() , v = read();
        cout << sum[u] + sum[v] - 2 * sum[ lca(u,v) ] << "\n";
    }
    return 0;
}
```

3.5.6 树上差分

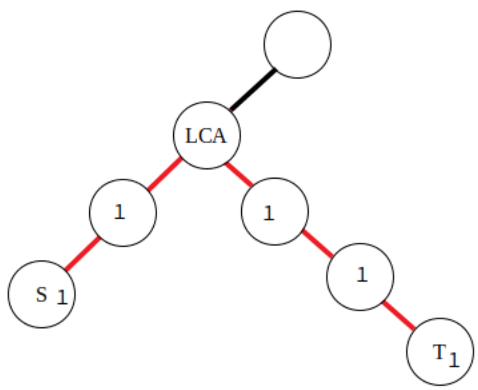
树上差分就是对树上某一段路径进行差分操作, 树上差分分为**点差分**和**边差分**

这里的差分数组用`d[i]`表示，其值是`a[fa[i]]-a[i]` **点差分**



点差分实际上是要对链分成两条链来操作，如果要对(S,T)路径上点权加p
则要d[s] += p , d[t] += p , d[lca] -= p , d[fa[lca]] -= p

边差分



因为直接对边差分比较困难，所以要把边权移动到边上的子节点上, 如果要对(S,T)路径上边权加p，则要d[s] += p , d[t] += p , d[lca] -= 2 * p

```
// Luogu P3128 边差分模板
#include<bits/stdc++.h>
using namespace std;

const int N = 5e4+5;
int n , m , fa[N][30] , logN , dep[N] ,p[N] , res;
vector<int> e[N];

int read(){...}

```

```
void dfs( int x ) {...} // 和 lca 的dfs完全相同

int lca( int x , int y ){...} // lca

void getRes( int x ){
    for( auto v : e[x] ){
        if( fa[x][0] == v ) continue;
        getRes(v) , p[x] += p[v];
    }
    res = max( res, p[x] );
}

int32_t main(){
    n = read() , m = read() , logN = log2(n)+1;
    for( int u , v , i = 1 ; i < n ; i ++ )
        u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
           );
    dep[1] = 1;
    dfs( 1 );
    for( int i = 1 , u , v , d ; i <= m ; i ++ ){
        u = read() , v = read() , d = lca( u , v );
        p[u] ++ , p[v] ++ , p[d] -- , p[ fa[d][0] ] --;
    }
    getRes( 1 );
    cout << res << "\n";
}
```

3.6 有向图连通性

3.6.1 强连通分量

连通在有向图中存在 u 到 v 的路径，则称 u 可达 v。如果 u,v 互相可达，则 u,v 连通。

强连通有向图 G 强连通指 G 中任意两个结点连通。

强联通分量有向图的极大强连通子图。

DFS 生成树

在有向图上进行 DFS 会形成森林。DFS 会形成 4 种边。

1. Tree Edge，树边

2. Back Edge，返祖边，指向祖先结点的边

3. Cross Edge，衡叉边，指向搜索过程中已经访问过的结点，但是这个结点并不是祖先节点

4. Forward Edge，前向边，指向子孙节点的边

如果结点 u 是某个强连通分量在搜索树中遇到的第一个结点，那么这个强连通分量的其余结点肯定是在搜索树中以 u 为根的子树中。结点 u 被称为这个强连通分量的根。

Tarjan

```
int cnt = 0, sc = 0; // cnt 记录 dfs 序 , sc 记录 强连通分量编号
stack<int> stk;
vector<int> dfn, low, inStk, scc, capacity;
// low[i] 点i所在子树的节点经过最多一条非树边能到达的结点中最小的dfs序
void tarjan(int x) {
    low[x] = dfn[x] = ++cnt;
    inStk[x] = 1, stk.push(x);
    for (auto y: e[x]) {
        if (!dfn[y])
            tarjan(y), low[x] = min(low[x], low[y]);
        else if (inStk[y])
            low[x] = min(low[x], dfn[y]);
    }
    if (low[x] == dfn[x]) {
        sc++, capacity.push_back(0);
        for (int y; true;) {
            y = stk.top(), stk.pop();
            inStk[y] = 0, scc[y] = sc, capacity[sc]++;
            if (x == y) break;
        }
    }
}

// 因为有向图搜索会形成森林
for (int i = 1; i <= n; i++)
    if (!dfn[i]) tarjan(i);
```

缩点

```
for( int x = 1 ; x <= n ; x ++ )
    for( auto y : e[x] )
        if( scc[x] != scc[y] ) g[scc[x]].push_back( scc[y] );
```

3.7 无向图连通性

3.7.1 割点

若对于 $x \in V$，从图中删除节点 x 以及所有 x 链接的边后，G 分裂成两个或两个以上个不相连的子图，则称 x 为 G 的**割点**

割点判定法则

若 x 不是搜索树的根节点，则 x 是割点当且仅当搜索树上存在一个 x 的子节点 y 满足 $dfn[x] \leq low[y]$.

特别的，若 x 是搜索树的根节点，则 x 是割点的条件当且仅当搜索树上存在至少两个子节点满足.

```
vector<vector<int>> e;

int cnt = 0;
vector<int> dfn, low;
vector<int> cut; // 储存所有的割点

void tarjan( int p , bool root = true ){
    int tot = 0;
    low[p] = dfn[p] = ++ cnt;
    for( auto q : e[p] ){
        if( !dfn[q] ){
            tarjan( q , false );
            low[p] = min( low[p] , low[q] );
            tot += ( low[q] >= dfn[p]); // 统计满足条件的子节点数
        }else low[p] = min( low[p] , dfn[q] );
    }
    if( tot > root ) cut.push_back(p);
    return ;
}
```

3.7.2 桥

若对于 $e \in E$，从图中删除边 e 之后，G 分裂成两个不相连的子图，则称 e 为 G 的**桥**或**割边**。

割边判定法则

无向边 (x, y) 是桥，当且仅当搜索树上存在 x 的一个子节点 y，满足 $dfn[x] \leq low[y]$.

桥一定是搜索树中的边，一个简单环中的边一定都不是桥。

```
vector<pii> bridges;
vector<vi> e;
```

```
vi dfn, low, fa;
int cnt;

void tarjan(int x) {
    low[x] = dfn[x] = ++cnt;
    for (auto y: e[x]) {
        if (!dfn[y]) {
            fa[y] = x, tarjan(y);
            low[x] = min(low[x], low[y]);
            if (low[y] > dfn[x])
                bridges.emplace_back(x, y);
        } else if (fa[x] != y)
            low[x] = min(low[x], dfn[y]);
    }
    return;
}
```

3.8 图论杂项

3.8.1 判断简单无向图图

根据图中的每个点的度可以判断，这个图是不是一个简单无向图（没有重边和自环的无向图）

Havel-Hakimi 定理

1. 对当前数列排序，使其呈递减
2. 从 S[2] 开始对其后 S[1] 个数字-1
3. 一直循环直到当前序列出现负数（即不可简单图化的情况）或者当前序列全为 0（可简单图化）时退出。

```
//NC-contest-38105-K
int read() {...}
priority_queue<int> q; vector<int> ve;
int32_t main() {
    int n = read();
    for( int i = 1 , x ; i <= n ; i ++ ){
        x = read() , q.push(x);
        if( x >= n ) // 如果度数大于 n 一定存在重边或自环
            cout << "NO\n" , exit(0);
```

```
    }
    while( 1 ){
        int k = q.top(); q.pop();
        if( k == 0 )
            cout << "YES\n" , exit(0);
        ve.clear();
        for( int x ; k ; k -- ){
            x = q.top() - 1 , q.pop();
            if( x < 0 )
                cout << "NO\n" , exit(0);
            ve.push_back(x);
        }
        for( auto it : ve ) q.push(it);
    }
    return 0;
}
```

3.8.2 2-SAT

SAT 是是适定性（Satisfiability）问题的简称。一般形式为 k-适定性问题，简称 k-SAT。而当 $k > 2$ 时该问题为 NP 完全的。

定义

有 n 个布尔变量 $x_1 \sim x_n$，另有 m 个需要满足的条件，每个条件的形式都是「x_i 为 `true/false` 或 x_j 为 `true/false`」。比如「x_1 为真或 x_3 为假」、「x_7 为假或 x_2 为假」。

$$(x_i x_j)(\neg x_i x_j)\cdots$$

注意这里的或为**排斥或**

2-SAT 问题的目标是给每个变量赋值使得所有条件得到满足。

Tarjan SCC 缩点

把每个变量拆成两个点，$X(True)$ 和 $X(False)$。比如现在有一个要求 $X|Y = True$，则把 $X(False)$ 到 $Y(True)$ 连一条边，把 $X(True)$ 到 $Y(False)$ 连一条边。连出来的图是对称的，然后跑一遍 Tarjan，如果存在某个变量的两个点在同一个强连通分量中的情况，则无解，否则有解。

构造方案时，我们可以通过缩点后的拓扑序确定变量的值。如果变量 x 的拓扑序在 $\neg x$ 之后，则取 x 为真，反之取 x 为假。注意 Tarjan 求得的 SCC 的编号相同与反拓扑序。

```
// luogu P4782
int32_t main() {
    int n, m, N;
    cin >> n >> m, N = n * 2 + 2;
```

```
    e.resize(N);
    dfn = inStk = low = scc = vector<int>(N);
    capacity.push_back(0);
    for (int i, a, j, b; m; m--) {
        cin >> i >> a >> j >> b;
        e[2 * i + (a ^ 1)].push_back(2 * j + b);
        e[2 * j + (b ^ 1)].push_back(2 * i + a);
    }
    for (int i = 2; i <= n * 2 + 1; i++)
        if (!dfn[i])
            tarjan(i);
    vector<int> res(n + 1);
    for (int i = 1; i <= n; i++) {
        if (scc[i * 2] == scc[i * 2 + 1])
            cout << "IMPOSSIBLE\n", exit(0);
        else if (scc[i * 2] > scc[i * 2 + 1])
            res[i] = 1;
    }
    cout << "POSSIBLE\n";
    for (int i = 1; i <= n; i++)
        cout << res[i] << " ";
    cout << "\n";
    return 0;
}
```

3.9 二分图

3.9.1 二分图

定义

给一张无向图，可以把点分成两个不相交的非空集合，并且在同一集合的点之间没有边相连，那么称这张无向图为一个二分图。

二分图的判定

一张无向图是二分图，当且仅当图中不存在奇环。

```
vector<vector<int>> e;
vector<int> v; // 会把所有的点染成 1 2 两种颜色
bool flag;

```

```
void dfs( int x , int color ){
    v[x] = color;
    for( auto y : e[x] ){
        if( v[y] == 0 ) dfs( y , 3 - color );
        else if( v[y] == color ){
            flag = false;
            return ;
        }
    }
}

bool check(){
    flag = true;
    for( int i = 1 ; flag and i <= n ; i ++ ){
        if( v[i] ) continue;
        dfs( i , 1 );
    }
    return flag;
}
```

3.9.2 二分图最大匹配

“任意两条边都没有公共端点”的边集合被称为一组的匹配。

在二分图中包含边数最多的一组匹配被称为二分图的最大匹配。

对于任意一组匹配 S（S 是一个边集）, 属于 S 的边叫匹配边，匹配边的端点叫匹配点。

如果在二分图中存在连接两个非匹配点的路径 $path$，则称 $path$ 是 S 的增广路。

增广路存在以下性质:

1. 长度是奇数
2. 路径上第 $1, 3, 5, \ldots$ 是非匹配边，第 $2, 4, 6, \ldots$ 是匹配边

所以对于一组匹配，把增广路上的边的状态全部取反，得到新的边集合 S’，则 S' 也是一组匹配，且匹配变数多一。

所以二分图的一组最大匹配 S，当且仅当二分图中不存在 S 的增广路。

匈牙利算法

算法流程

1. 设 $S = \emptyset$
2. 求增广路 $path$，把路径上边的状态取反得到新的匹配 S'

3. 重复第 2 步直到没有增广路

代码实现采用深搜，从 x 出发寻找增广路，并且还回溯时把状态取反。

N 个点 M 条的二分图，复杂度 $O(NM)$

```
// luogu P3386
//给定一个二分图，其左侧点n个，右侧点的个数为m，边数为k，求其最大匹配的边数。
//左侧点编号[1,n]，右侧点编号[1,m]。
#include <bits/stdc++.h>

using namespace std;

int n, m, k; // 左右两个集合的元素数量
vector<vector<int>> e;
vector<int> p; // 当前右侧集合对于左侧的元素
vector<bool> vis;// 右侧元素是否已经被访问过

bool match(int x) {
    for (auto y: e[x]) {
        if (vis[y]) continue;
        vis[y] = 1;
        if (p[y] == 0 or match(p[y])) { // 暂未匹配或原来匹配的左侧元素可以找到新的匹配
            p[y] = x;
            return true;
        }
    }
    return false;
}

int Hungarian() {
    int cnt = 0;
    p = vector<int>(n + m + 1);
    for (int i = 1; i <= n; i++) { // 枚举左侧元素
        vis = vector<bool>(n + m + 1);
        if (match(i)) cnt++;
    }
    return cnt;
}
```

```

int32_t main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n >> m >> k;
    e = vector<vector<int>>(n + m + 1);

    for (int x, y; k; k--) {
        cin >> x >> y;
        y += n; // 为了方便表示把右侧点映射为 [n+1,n+m]
        e[x].push_back(y), e[y].push_back(x);
    }
    cout << Hungarian() << "\n";
    return 0;
}
```

3.9.3 二分图的最小的点覆盖

给一张二分图，求最小点集 S，使得图中任意一条边都至少有一个端点属于 S。

König 定理

二分图最小点覆盖包含的点数等于二分图最大匹配包含的边数。

构造方法

1. 求出二分图的最大匹配
2. 从左部每个非匹配点出发，再执行一次 DFS 求增广路的过程（一定会失败），标记访问过的所有点
3. 取左部未被标记的点、右部被标记的点，就得到了最小点覆盖。

3.9.4 二分图的最大独立集

给一张无向图，图的独立集就是任意两点之间没有边相连的点集，包含点数最多的独立集就是图的最大独立集。

任意两点之间都有一条边相连的子图被称作无向图的团，点数最多团就是图的最大团。

对于一般的无向图，最大团、最大独立集是 NP 完全问题。

定理

无向图的最大团就是补图的最大独立集

定理

G 是 n 个点的二分图，G 的最大独立集大小等于 n 减去最大匹配数。

对于二分图去掉最小点覆盖，剩余的点就构成了二分图的最大独立集。

第四章　数学知识

4.1　数论

4.1.1　整除

定义：若整数 b 除以非零整数 a ，商为整数且余数为零我们就说 b 能被 a 整除，或 a 整除 b 记作 $a|b$

性质

1. 传递性，若 $a|b,b|c$，则 $a|c$

2. 组合性，若 $a|b,a|c$ 则对于任意整数 m,n 均满足 $a|mb+nc$

3. 自反性，对于任意的 n，均有 $n|n$

4. 对称性，若 $a|b,b|a$ 则 $a=b$

4.1.2　约数

定义若整数 n 除以整数 x 的余数为 0，即 d 能整除 n，则称 d 是 n 的约数，n 是 d 的倍数，记为 $d|n$ 算数基本定理由算数基本定理得正整数 N 可以写作 $N=p_1^{C_1}\times p_2^{C_2}\times p_3^{C_3}\cdots\times p_m^{C_m}$

分解质因数

分解成 $p_1\times_2\times p_3\times\cdots\ p_n$ 这种形式

```
vector< int > factorize( int x ){
    vector<int> ans;
    for( int i = 2 ; i * i <= x ; i ++){
        while( x % i == 0 )
            ans.push_back(i) , x /= i;
    }
    if( x > 1 ) ans.push_back(x);
    return ans;
}
```

分解成 $p_1^{k_1}\times p_2^{k_2}\times p_3^{k_3}\times\cdots\ p_n^{k_n}$

```
vector< pair<int,int> > factorize( int x ){
    vector<pair<int,int>> ans;
    for( int i = 2 , cnt ; i * i <= x ; i ++){
        if( x % i ) continue;
        cnt = 0;
        while( x % i == 0 ) cnt ++ , x /= i;
        ans.push_back( { i , cnt } );
    }
    if( x > 1 ) ans.push_back( { x , 1 } );
    return ans;
}
```

N 的正约数个数为 (Π 是连乘积的符号，类似 $\sum$)

$$(c_1+1)\times(c_2+1)\times\cdots(c_m+1)=\Pi_{i=1}^{m}(ci+1)$$

N 的所有正约数和为

$$(1+p_1+p_1^2+\cdots+p_1^{c_1})\times\cdots\times(1+p_m+p_m^2+\cdots+p_m^{c_m})=\prod_{i=1}^{m}(\sum_{j=0}^{c_i}(p_i)^j)$$

4.1.3 GCD 和 LCM

性质 $a\times b=gcd(a,b)\times(a,b)$

通过性质可以得到最小公倍数的求法就是

```
int lcm( int x , int y ){
    return a / gcd( x , y ) * b;
}
```

最大公倍数的求法有**更相减损术**和**辗转相除法**

```
int gcd( int x , int y ){ // 更相减损术
    while( x != y ){
        if( x > y ) x -= y;
        else y -=x;
    }
    return x;
}

int gcd( int x , int y ){ // 辗转相除法
    return b ? gcd( b , a % b ) : a ;
}
```

```

// 还有一种是直接调用库函数
__gcd( a , b );
```

一般情况下直接用库函数，库函数的实现是辗转相除法，如果遇到高精度的话（高精度取模分困难）可以用更相减损术来代替

定理对于斐波那契数列 Feb_i 有 $Feb_{gcd(a,b)} = gcd(Feb_a, Feb_b)$

4.1.4 质数

判断质数

```
bool isPrime( int x ){
    if( x < 3 or x % 2 == 0 ) return x == 2;
    for( int i = 2 ; i * i <= x ; i ++ )
        if( x % i == 0 ) return 0;
    return 1;
}
```

埃式筛

```
vector< int > prime;
bitset<N>notPrime;//不是素数

void getPrimes( int n ){
    notPrime[1] = notPrime[0] = 1;
    for( int i = 2 ; i <= n ; i ++ ){
        if( notPrime[i] ) continue;
        prime.push_back(i);
        for( int j = i * 2 ; j <= n ; j += i )
            notPrime[j] = 1;
    }
}

// 如果不需要 prime 数组的话可以优化成下面的代码
bitset<N>notPrime;//不是素数

void getPrimes( int n ){
    notPrime[1] = notPrime[0] = 1;
    for( int i = 2 ; i * i <= n ; i ++ ){
        if( notPrime[i] ) continue;
        for( int j = i * 2 ; j <= n ; j += i )
```

```
            notPrime[j] = 1;
    }
}
```

欧拉筛

```
vector< int > prime;
bool notPrime[N];;//不是素数

void getPrimes( int n ){
    notPrime[1] = notPrime[0] = 1;
    for( int i = 2 ; i <= n ; i ++ ){
        if( !notPrime[i] ) prime.push_back(i);
        for( auto it : prime ){
            if( it * i > n ) break;
            notPrime[ it * i ] = 1;
            if( i % it == 0 ) break;
        }
    }
}
```

证明质数有无限个

反证法假设数是 n 个，每个素数是 p_i，令 $P = \Pi_{i=1}^{n} p_i + 1$

因为任何一个数都可以分解成多个质数相乘

所以 P 除以任何一个质数都余 1 ，显然 P 就也是一个质数，与假设矛盾，所以假设错误

所以质数是无限个

性质 2

设 $\pi(n)$ 为不超过 n 的质数个数，则 $\pi(n) \approx \frac{n}{\ln n}$

4.1.5 逆元

费马小定理

$a^{p-1} \equiv 1(mod\ p)$, 其中 p 为素数, 所以 $aa^{p-2} \equiv 1(mod\ p)$

```
int inv( int x ) {return pow( x , p - 2 );}
```

$O(n)$ **递推**

```
const int N = 1005;
int inv[N] = {};
void invers(int n,int mod){
    inv[1] = 1;
    for(int i = 2;i <= n;i ++) inv[i] = (p-p/i) * inv[p%i] % p;
```

```
    return ;
}
```

4.1.6 扩展欧几里得

裴蜀定理

设 a,b 是不全为零的整数，则存在整数 x,y，使得 $ax+by=\gcd(a,b)$

```
int exgcd( int a , int b , int & x , int & y ){
    if( b == 0 ) { x = 1 , y = 0 ; return a;}
    int d = exgcd( b , a%b , x , y );
    int z = x ; x = y ; y = z - y * (a / b);
    return d;
}

template<typename T>
T exgcd(T a, T b, T &x, T &y){
    if (!b){
        x = 1, y = 0;
        return a;
    }
    T r = exgcd(b, a % b, y, x);
    y -= a / b * x;
    return r;
}
```

丢番图方程

$ax+by=c$

定义变量 d,x_0,y_0，调用`d = exgcd(a, b, x0, y0)`。对于方程的特解为

$$(x=\frac{c}{d}x_0,y=\frac{c}{d}y_0)$$

对于方程的通解为

$$(x=\frac{c}{d}x_0+k\frac{b}{d},y=\frac{c}{d}y_0-k\frac{a}{d}),k\in Z$$

线性同余方程

$a\times x\equiv b(\mod m)$

线性同余方程等价于 $a\times x-b$ 是 m 的倍数，设为 $-y$ 倍，方程可改写为丢番图方程 $a\times x+m\times y=b$

线性同余方程有解的充要条件 $\gcd(a,m)|b$

在有解时用扩偶求得 x_0,y_0 满足 $a\times x_0+m\times y_0=\gcd(a,m)$，则方程的特解 $x=x_0\times\frac{b}{\gcd(a,m)}$

通解是 $x=x_0\times\frac{b}{\gcd(a,m)}+k\times\frac{m}{\gcd(a,m)},k\in Z$

```
int calc(int a, int b, int m) {
    int x, y, d;
    d = exgcd(a, m, x, y);
    if (b % d) return -1;
    return x * b / d;
}
```

扩展欧几里得求逆元

本质上是解同余方程 $a \times a^{-1} \equiv 1 \pmod m$

```
int inv(int a, int m) {
    int x, y, d;
    d = exgcd(a, m, x, y);
    if (d != 1) return -1;
    return (x % m + m) % m;
}
```

线性同余方程组（中国剩余定理）

设 $m_1, m_2, \ldots, m_n$ 是两两互质的整数。$m = \Pi_{i=1}^{n} m_i, M_i = \frac{m}{m_i}$，$t_i$ 是线性同余方程组 $M_i t_i \equiv 1 \pmod m$ 的一个解，对于任意的 n 个整数 $a_1, a_2, \ldots, a_n$，方程组

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

有整数解，解为 $x = \sum_{i=1}^{n} a_i M_i t_i \pmod m$

```
int CRT(int n, vector<int> a, vector<int> m) {
    int mm = 1, ans = 0;
    for (int i = 1; i <= n; i++) mm = mm * m[i];
    for (int i = 1; i <= n; i++) {
        int M = mm / m[i], t, y;
        exgcd(M, m[i], t, y); // t * M % m[i] = 1;
        ans = (ans + a[i] * M * t % mm) % mm;
    }
    return ans;
}
```

高次同余方程 (BSGS)

$a^x \equiv b \pmod p$，要求 a, p 互质。求非负整数 x。复杂度 $O(\sqrt{p})$

```
int BSGS(int a, int b, int p) {
    map<int, int> hash;
```

```
    b %= p;
    int t = sqrt(p) + 1;
    for (int j = 0, val; j < t; j++) {
        val = b * power(a, j, p) % p;
        hash[val] = j;
    }
    a = power(a, t, p);
    if (a == 0) return b == 0 ? 1 : -1;
    for (int i = 0, val, j; i <= t; i++) {
        val = power(a, i, p);
        j = hash.find(val) == hash.end() ? -1 : hash[val];
        if (j >= 0 && i * t - j >= 0) return i * t - j;
    }
    return -1;
}
```

4.2 数学杂项

4.2.1 二维向量的叉积

$\vec{A} \times \vec{B} = |A||B|\cos(\alpha) = a_x \times b_y - a_y \times b_x$

虽然叉积的结果是一个标量，但是叉积是有正负的，正负取决于向量夹角的大小。

应用：判断点是否在三角形的内部对于三角形 abc 和一个点 o，$\vec{ao} \times \vec{ab}$ 的正负表示了点 o 在线 ab 的左侧还是右侧。只要按照顺时针方向（或逆时针方向）判断点 o 在三条直线的同一侧，既可以判断点在三角形的内部。

```
class Triangle{
    typedef std::pair<int,int> Vector;
private:
    Vector getVector( int x , int y , int a , int b ){
        return std::pair{ a - x , b - y };
    }
    bool product( Vector a , Vector b ){
        return ( a.first * b.second - a.second * b.first ) > 0;
    }
public:
    int ax , ay , bx ,by , cx , cy;
    Vector ab , bc , ca;
    Triangle( int ax , int ay , int bx , int by , int cx , int cy ):
```

```
            ax(ax) , ay(ay) , bx(bx) , by(by) , cx(cx) , cy(cy) ,
            ab( getVector( ax , ay , bx , by ) ),
            bc( getVector( bx , by , cx , cy ) ),
            ca( getVector( cx , cy , ax , ay ) ){};
    Triangle(){};
    bool isInTriangle( int x , int y ){
        Vector ao = getVector( ax , ay , x , y );
        Vector bo = getVector( bx , by , x , y );
        Vector co = getVector( cx , cy , x , y );
        bool f1 = product( ao , ab ) , f2 = product( bo , bc ) , f3 =
           product( co , ca );
        return ( f1 == f2 && f2 == f3 );
    }
};
```

4.3 组合数学

4.3.1 公式

排列数

$$A_n^n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$$

组合数

$$C_m^n = \frac{m!}{(m-n)! \times n!}$$

组合数性质

1. $C_n^m = C_n^{n-m}$
2. $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
3. $C_n^0 + C_n^1 + \cdots + C_n^n = 2^N$
4. $C_n^0 + C_n^2 + C_n^4 + \cdots = C_n^1 + C_n^3 + C_n^5 \cdots = 2^{n-1}$
5. $C_n^m = \frac{n-m+1}{n} \times C_n^{m-1}$

4.3.2 组合数计算

```
// 暴 力 计 算 组 合 数
int C(int x, int y) { // x 中 选 y 个
    y = min(y, x - y);
```

```
    int res = 1;
    for (int i = x, j = 1; j <= y; i--, j++)
        res = res * i / j;
    return res;
}

// 加法递推O(n^2)
for( int i = 0 ; i <= n ; i ++ ){
    c[i][0] = 1;
    for( int j = 1 ; j <= i ; j ++)
        c[i][j] = c[i-1][j] + c[i-1][j-1];
}
// 乘法递推O(n)
c[0] = 1;
for( int i = 1 ; i * 2 <= n ; i ++ )
    c[i] = c[n-i] = ( n-i+1 ) * c[i-1] / i;

// 任意组合数
#define int long long
const int N = 5e5+5 , mod = 1e9+7;
int fact[N] , invFact[N];

int power(int x,int y){...} // 快速幂
int inv( int x ){...} // 求逆元，一般是费马小定理

int A( int x , int y ){ // x 中选 y 排序
    return fact[x] * invFact[x-y] % mod;
}

int C( int x , int y ){ // x 中选 y 个
    return fact[x] * invFact[x-y] % mod * invFact[y] % mod;
}

void init(){
    fact[0] = 1 , invFact[0] = inv(1);
    for( int i = 1 ; i < N ; i ++ )
        fact[i] = fact[i-1] * i % mod , invFact[i] = inv(fact[i]);
}
```

4.3.3 公式杂项

$1^2+2^2+3^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}$

4.4 线性代数

4.4.1 矩阵加速递推

```
struct matrix {
    static constexpr int mod = 1e9 + 7;
    int x, y;
    vector<vector<int>> v;

    matrix() {}

    matrix(int x, int y) : x(x), y(y) {
        v = vector<vector<int>>(x + 1, vector<int>(y + 1, 0));
    }

    void I() {// 单位化
        y = x;
        v = vector<vector<int>>(x + 1, vector<int>(x + 1, 0));
        for (int i = 1; i <= x; i++) v[i][i] = 1;
        return;
    }

    void display() { // 打印
        for (int i = 1; i <= x; i++)
            for (int j = 1; j <= y; j++)
                cout << v[i][j] << " \n"[j == y];
        return;
    }

    friend matrix operator*(const matrix &a, const matrix &b) { //乘法
        assert(a.y == b.x);
        matrix ans(a.x, b.y);
        for (int i = 1; i <= a.x; i++)
            for (int j = 1; j <= b.y; j++)
                for (int k = 1; k <= a.y; k++)
```

```
                    ans.v[i][j] = (ans.v[i][j] + a.v[i][k] * b.v[k][j]) % mod;
        return ans;
    }

    friend matrix operator^( matrix x , int y ){ // 快速幂
        assert( x.x == x.y );
        matrix ans(x.x , x.y);
        ans.I();//注意一定要先单位化
        while( y ){
            if( y&1 ) ans = ans*x;
            x = x * x , y >>= 1;
        }
        return ans;
    }
};
```

例题 Luogo P1939

已知数列 a，满足

$$a_i = \begin{cases} 1 & i \in \{1,2,3\} \\ a_{i-1} + a_{x-3} & x \geq 4 \end{cases}$$

求数列第 n 项对 10^9+7 取模

设计状态阵 $mat_i = \begin{bmatrix} a_i \\ a_{i-1} \\ a_{i-2} \end{bmatrix}$，则 $mat_{i+1} = \begin{bmatrix} a_{i+1} \\ a_i \\ a_{i-1} \end{bmatrix} = \begin{bmatrix} a_i + a_{i-2} \\ a_i \\ a_{i-1} \end{bmatrix}$

可以用待定系数法，加对应项相等解出转移矩阵 $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

则 $mat_{3+n} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^n \times mat_3$

4.5 离散数学

4.6 计算几何

4.6.1 基础模板

```
/* 板子来自 Pecco 采取以下原则
```

```
 * 写全局函数，而非类方法，结构体只存储数据
 * 每个函数标注以来那些函数，且尽量减少依赖
 * 用较为简略的名字，同时传值而非 const 引用
 * */

#define mp make_pair
using db = double; //精度不够时可换 long double
// 几何对象
struct Point {// 点
    db x, y;

    Point(db x = 0, db y = 0) : x(x), y(y) {};
};

using Vec = Point;// 向量
struct Line { // 直线 (点向式)
    Point P;
    Vec v;

    Line(Point P, Vec v) : P(P), v(v) {};
};

struct Seg { // 线段(存两个端点)
    Point A, B;

    Seg(Point A, Point B) : A(A), B(B) {};
};

struct Circle { // 圆(存圆心和半径)
    Point O;
    db r;

    Circle(Point O, db r) : O(O), r(r) {};
};

// 常用常量
const Point O(0, 0); // 原点
const Line Ox(O, Vec(1, 0)), Oy(O, Vec(0, 1));// 坐标轴
```

```
const double pi = M_PI, eps = 1e-9;// pi 精度最高的是acosl(-1)

// 浮点比较
bool eq(db a, db b) { return abs(a - b) < eps; }// ==
bool gt(db a, db b) { return a - b > eps; }     // >
bool lt(db a, db b) { return a - b < -eps; }    // <
bool ge(db a, db b) { return a - b > -eps; }    // >=
bool le(db a, db b) { return a - b < eps; }     // <=

// 基础操作
Vec r90a(Vec v) { return Vec(-v.y, v.x); }// 顺时针旋转 90 度
Vec r90c(Vec v) { return Vec(v.y, -v.x); }// 逆时针旋转 90 度
// 向量加法
Vec operator+(Vec u, Vec v) { return Vec(u.x + v.x, u.y + v.y); }

// 向量减法
Vec operator-(Vec u, Vec v) { return Vec(u.x - v.x, u.y - v.y); }

// 数乘
Vec operator*(db k, Vec v) { return Vec(k * v.x, k * v.y); }

// 点乘
db operator*(Vec u, Vec v) { return u.x * v.x + u.y * v.y; }

// 叉乘
db operator^(Vec u, Vec v) { return u.x * v.y - u.y * v.x; }

db len(Vec v) { return sqrt(v.x * v.x + v.y * v.y); }// 向量长度
db slope(Vec v) { return v.y / v.x; }// 斜率

// 向量相关
// 两向量夹角余弦 u->v depends len,V*V
double cos_t(Vec u, Vec v) { return u * v / len(u) / len(v); }

//单位化 depends len
Vec norm(Vec v) { return Vec(v.x / len(v), v.y / len(v)); }

// 与原向量平行且横坐标大于等于 0 的单位向量 depends k*V , len , norm;
```

```
Vec pnorm(Vec v) {
    return (v.x < 0 ? -1 : 1) * norm(v);
    return (v.x < 0 ? -1 : 1) / len(v) * v;// 不依赖 norm
}

// 线段的方向向量 depends V-V // 直线的方向向量直接访问 v
Vec dvec(Seg l) { return l.B - l.A; }

// 直线相关
// 两点式求直线 depends V-V
Line line(Point A, Point B) { return Line(A, B - A); }

// 斜截式求直线
Line line(db k, db b) { return Line(Point(0, b), Vec(1, k)); }

// 点斜式求直线
Line line(Point P, db k) { return Line(P, Vec(1, k)); }

// 线段所在直线 depends V-V
Line line(Seg l) { return Line(l.A, l.B - l.A); }

db at_x(Line l, db x) {// 给定直线的横坐标求纵坐标
    assert(l.v.x != 0);
    return l.P.y + (x - l.P.x) * l.v.y / l.v.x;
}
db at_y(Line l, db y) {// 给定直线的纵坐标求横坐标
    assert(l.v.y != 0);
    return l.P.x + (y - l.P.y) * l.v.x / l.v.y;
}

Point pedal(Point P, Line l) {// 点到直线的垂足 depends V-V, V*V, d*V
    return l.P - (l.P - P) * l.v / (l.v * l.v) * l.v;
}

// 过某点作直线的垂线 depends r90c
Line perp(Line l, Point P) { return Line(P, r90c(l.v)); }

Line bisec(Point P, Vec u, Vec v) {// 角平分线 depends V+V, len, norm
```

```
    return Line(P, norm(u) + norm(v));
}

// 线段相关
Point midp(Seg l) { // 线段中点
    return Point((l.A.x + l.B.x) / 2, (l.A.y + l.B.y) / 2);
}
Line perp(Seg l) { // 线段中垂线 depends r90c, V-V, midp
    return Line(midp(l), r90c(l.B - l.A));
}

// 几何对象之间的关系
// 向量是否垂直 depends eq, V*V
bool verti(Vec u, Vec v) { return eq(u * v, 0); }

// 向量是否平行 depends eq, V^V
bool paral(Vec u, Vec v) { return eq(u ^ v, 0); }

// 向量是否与x轴平行 depends eq
bool paral_x(Vec v) { return eq(v.y, 0); }

// 向量是否与y轴平行 depends eq
bool paral_y(Vec v) { return eq(v.x, 0); }

bool on(Point P, Line l) {// 点是否在直线上 depends eq
    return eq((P.x - l.P.x) * l.v.y, (P.y - l.P.y) * l.v.x);
}

bool on(Point P, Seg l) { // 点是否在线段上 depends eq, len, V-V
    return eq(len(P - l.A) + len(P - l.B), len(l.A - l.B));
}

bool operator==(Point A, Point B) { // 两点是否重合 depends eq
    return eq(A.x, B.x) and eq(A.y, B.y);
}

bool operator==(Line a, Line b) { // 两条直线是否重合 depends eq,on(L)
    return on(a.P, b) and on(a.P + a.v, b);
```

```
}

bool operator==(Seg a, Seg b) { // 两条线段是否重合 depends eq, P==P
    return (a.A == b.A and a.B == b.B) or (a.A == b.B and a.B == b.A);
}

// 以横坐标为第一关键字、纵坐标为第二关键字比较两个点 depends eq, lt
bool operator<(Point A, Point B) {
    return lt(A.x, B.x) or (eq(A.x, B.x) and lt(A.y, B.y));
}

// 直线与圆是否相切 depends eq, V^V, len
bool tangency(Line l, Circle c) {
    return eq(abs((c.O ^ l.v) - (l.P ^ l.v)), c.r * len(l.v));
}

// 圆与圆是否相切 depends eq, V-V, len
bool tangency(Circle C1, Circle C2) {
    return eq(len(C1.O - C2.O), C1.r + C2.r);
}

// 距离
// 两点之间的距离 depends len, V-V
db dis(Point A, Point B) { return len(A - B); }

// 点到直线的距离 depends V^V, len
db dis(Point P, Line l) {
    return abs((P ^ l.v) - (l.P ^ l.v)) / len(l.v);
}

// 平行直线间的距离 depends d*V, V^V, len, pnrom
db dis(Line a, Line b) {
    assert(paral(a.v, b.v));
    return abs((a.P ^ pnorm(a.v)) - (b.P ^ pnorm(b.v)));
}

// 平移和旋转
// 平移 depends V+V
```

```
Line operator+(Line l, Vec v) { return Line(l.P + v, l.v); }
Seg operator+(Seg l, Vec v) { return Seg(l.A + v, l.B + v); }

// 旋转 depends V+V, V-V
Point rotate(Point P, db rad) {
    return Point(cos(rad)*P.x-sin(rad)*P.y,sin(rad)*P.x+cos(rad)*P.y);
}
Point rotate(Point P, db rad, Point C) {
    return C + rotate(P - C, rad);
}

Line rotate(Line l, db rad, Point C = O) {
    return Line(rotate(l.P, rad, C), rotate(l.v, rad));
}

Seg rotate(Seg l, db rad, Point C = O) {
    return Seg(rotate(l.A, rad, C), rotate(l.B, rad, C));
}

// 对称
// 关于点的对称
Point reflect(Point A, Point P) {
    return Point(P.x * 2 - A.x, P.y * 2 - A.y);
}

Line reflect(Line l, Point P) { return Line(reflect(l.P, P), l.v); }

Seg reflect(Seg l, Point P) {
    return Seg(reflect(l.A, P), reflect(l.B, P));
}

// 关于直线对称 depends V-V, V*V, d*V , pedal
Point reflect(Point A, Line ax) { return reflect(A, pedal(A, ax)); }

Vec reflect_v(Vec v, Line ax) {
    return reflect(v, ax) - reflect(O, ax);
}

```

```
Line reflect(Line l, Line ax) {
    return Line(reflect(l.P, ax), reflect_v(l.v, ax));
}

Seg reflect(Seg l, Line ax) {
    return Seg(reflect(l.A, ax), reflect(l.B, ax));
}
// 交点
// 直线与直线交点 depends eq, d*V, V*V, V+V, V^V
vector <Point> inter(Line a, Line b) {
    vector <Point> ans;
    double c = a.v ^ b.v;
    if (eq(c, 0)) return ans;
    Vec v = 1 / c * Vec(a.P ^ (a.P + a.v), b.P ^ (b.P + b.v));
    ans.emplace_back(v * Vec(-b.v.x, a.v.x), v * Vec(-b.v.y, a.v.y));
    return ans;
}

// 直线与圆交点 depends eg, gt, V+V, V-V, V*V, d*V, len, pedal
vector <Point> inter(Line l, Circle C) {
    vector <Point> ans;
    Point P = pedal(C.O, l);
    double h = len(P - C.O);
    if (gt(h, C.r)) return ans;
    if (eq(h, C.r)) {
        ans.emplace_back(P);
        return ans;
    }
    double d = sqrt(C.r * C.r - h * h);
    Vec vec = d / len(l.v) * l.v;
    ans.emplace_back(P + vec);
    ans.emplace_back(P - vec);
    return ans;
}

// 圆与圆的交点 depends eq, gt, V+V, V-V, d*V, len , r90c
vector <Point> inter(Circle C1, Circle C2) {
    vector <Point> ans;
```

```
    Vec v1 = C2.O - C1.O, v2 = r90c(v1);
    double d = len(v1);
    if (gt(d, C1.r + C2.r) || gt(abs(C1.r - C2.r), d)) return ans;
    if (eq(d, C1.r + C2.r) || eq(d, abs(C1.r - C2.r))) {
        ans.emplace_back(C1.O + C1.r / d * v1);
        return ans;
    }
    double a = ((C1.r * C1.r - C2.r * C2.r) / d + d) / 2;
    double h = sqrt(C1.r * C1.r - a * a);
    Vec av = a / len(v1) * v1, hv = h / len(v2) * v2;
    ans.emplace_back(C1.O + av + hv),ans.emplace_back(C1.O + av - hv);
    return ans;
}

// 三角形的四心
// 三角形重心
Point baryCenter(Point A, Point B, Point C) {
    return Point((A.x + B.x + C.x) / 3, (A.y + B.y + C.y) / 3);
}

// 三角形外心 depends r90c, V*V, d*V, V-V, V+V
// 给定三点求圆, 要先判断是否三点共线
Point circumCenter(Point A, Point B, Point C) {
    double a = A * A, b = B * B, c = C * C;
    double d = 2 * (A.x*(B.y-C.y) + B.x*(C.y-A.y) + C.x*(A.y-B.y));
    return 1 / d * r90c(a * (B - C) + b * (C - A) + c * (A - B));
}

// 三角形内心 depends len, d*V, V-V, V+V
Point inCenter(Point A, Point B, Point C) {
    double a = len(B - C), b = len(A - C), c = len(A - B);
    double d = a + b + c;
    return 1 / d * (a * A + b * B + c * C);
}

// 三角形垂心 depends V*V, d*V, V-V, V^V, r90c
Point orthoCenter(Point A, Point B, Point C) {
    double n = B * (A - C), m = A * (B - C);
```

```
    double d = (B - C) ^ (A - C);
    return 1 / d * r90c(n * (C - B) - m * (C - A));
}

// 两圆相交的面积 depends V-V, dis
ldb areaInter(Circle c1, Circle c2) {
    ldb d = dis(c1.o, c2.o);
    auto &r1 = c1.r, r2 = c2.r;
    if (d >= r1 + r2) return 0.0;
    if (r1 + d <= r2) return Pi * r1 * r1;
    if (r2 + d <= r1) return Pi * r2 * r2;
    ldb ans = 0;

    ldb ang1 = acosl((r1 * r1 + d * d - r2 * r2) / (2 * r1 * d));
    ans += ang1 * r1 * r1;
    ans -= r1 * sinl(ang1) * r1 * cosl(ang1);

    ldb ang2 = acosl((r2 * r2 + d * d - r1 * r1) / (2 * r2 * d));
    ans += ang2 * r2 * r2;
    ans -= r2 * sinl(ang2) * r2 * cosl(ang2);
    return ans;
}
```

4.6.2 极角序

直接计算极角, atan2(y,x)函数可直接计算(x,y)的极角, 值域是 $(-\pi,\pi]$。注意第四象限的极角比第一象限要小。

```
db theta(Vec v) {
    return atan2(v.y, v.x);
}

void psort(Points &ps, Point c = O) {
    sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
        return theta(p1 - c) < theta(p2 - c);
    });
}
```

先比较象限再做叉乘

```
int qua(Point p) { // 求象限
```

```
    return lt(p.y, 0) << 1 | lt(p.x, 0) ^ lt(p.y, 0);
}

void psort(Points &ps, Point c = O) {
    sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
        return qua(p1-c) < qua(p2-c) || (qua(p1-c) == qua(p2-c) && lt((p1-c) ^ (p2-c), 0));
    });
}
```

这种方法常数可能稍微大一点，但是精度比较好，如果坐标都是整数的话是完全没有精度损失的。

4.6.3 凸包

Graham 扫描法

最左下角的一个点，一定在凸包上，以这个角为极点，进行极角排序，然后逐个点扫描。用栈来维护，如果栈中点数小于 3，就直接进栈；否则，检查栈顶三个点组成的两个向量的旋转方向是否为逆时针（这可以用叉乘判断），若是则进栈，若不是则弹出栈顶，直到栈中点数小于 3 或者满足逆时针条件为止。

实现时需要注意，要对极角排序的极点特殊处理，使它始终排在第一位。

```
// depends eq , lt , cross , V-V , P < P , len
using Points = vector<Point>;

bool check(Point p, Point q, Point r) { // 检查是向量旋转方向是否为逆时针
    return lt(0, (q - p) ^ (r - q));
}

Points chull(Points &ps) {
    sort(ps.begin(), ps.end());
    vector<int> I{0}, used(ps.size());

    for (int i = 1; i < ps.size(); i++) {
        while (I.size() > 1 and !check(ps[I[I.size() - 2]], ps[I.back()], ps[i]))
            used[I.back()] = 0, I.pop_back();
        used[i] = 1, I.push_back(i);
    }
    for (int i = ps.size() - 2, tmp = I.size(); i >= 0; i--) {
```

```
        if (used[i]) continue;
        while (I.size() > tmp and !check(ps[I[I.size() - 2]], ps[I.
            back()], ps[i]))
            I.pop_back();
        used[i] = 1, I.push_back(i);
    }
    Points H;
    I.pop_back();
    for (auto i: I)
        H.push_back(ps[i]);
    return H;
}
```

Andrew 算法

另一种方法是不做极角排序，直接以横坐标为第一关键词、纵坐标为第二关键词排序，这样将顶点依次相连（不首尾相连）的话，也能保证不交叉。

然后同上一种方法类似，正反遍历两遍，分别求出上下凸包结合起来就好。

```
// depends eq , lt , cross , V-V , P < P , len
using Points = vector<Point>;

db theta(Point p) { // 极角
    return p == O ? -1 / 0. : atan2(p.y, p.x);
}

void psort(Points &ps, Point c = O) { // 极角序,先按照极角排序,再按照
    距离排序
    sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
        auto t1 = theta(p1 - c), t2 = theta(p2 - c);
        if (eq(t1, t2)) return lt(len(p1 - c), len(p2 - c));
        return lt(t1, t2);
    });
}

bool check(Point p, Point q, Point r) { // 检查是向量旋转方向是否为逆
    时针
    return lt(0, (q - p) ^ (r - q));
}

Points chull(Points &ps) {
```

```
    psort(ps, *min_element(ps.begin(), ps.end()));
    Points H{ps[0]};
    for (int i = 1; i < ps.size(); i++) {
        while (H.size() > 1 and !check(H[H.size() - 2], H.back(), ps[i
           ]))
            H.pop_back();
        H.push_back(ps[i]);
    }
    return H;
}
```

这种方法一般会比第一种快一点。

第五章 字符串

5.1 字符串哈希

5.1.1 单哈希

这里用unsigned long long的自然溢出做模数

```
typedef unsigned long long ull;
const int N = 1e6+5 , K = 1e9+7;// 长度 K进制
vector<ull> hashP(N);

void init(){// 初始化
    hashP[0] = 1;
    for( int i = 1 ; i < N ; i ++ ) hashP[i] = hashP[i-1] * K;
}

void hashStr( const string & s , vector<ull> & a){ // 计算Hash数组
    a.resize(s.size()+1);
    for( int i = 1 ; i <= s.size() ; i ++ )
        a[i] = ull(a[i-1] * K + s[i-1]);
}

ull hashStr( const string & s ){
    ull ans = 0;
    for( auto i : s )
        ans = ull( ans * K + i );
    return ans;
}

ull getHash( int l , int r , const vector<ull> & a ){ //计算Hash值
    return a[r] - a[l-1] *  hashP[r-l+1];
}
```

5.1.2 双哈希

这里用unsigned long long的自然溢出和998244353做模数

```
namespace Hash {// 下标从 1 开始
    using Val = pair<i64, i64>;
    const Val Mod(1e9 + 7, 1e9 + 9);
    const Val base(13331, 23333);
    vector <Val> p;

    Val operator+(Val a, Val b) {
        i64 c1 = a.first + b.first, c2 = a.second + b.second;
        if (c1 >= Mod.first) c1 -= Mod.first;
        if (c2 >= Mod.second) c2 -= Mod.second;
        return pair(c1, c2);
    }

    Val operator-(Val a, Val b) {
        i64 c1 = a.first - b.first, c2 = a.second - b.second;
        if (c1 < 0) c1 += Mod.first;
        if (c2 < 0) c2 += Mod.second;
        return pair(c1, c2);
    }

    Val operator*(Val a, Val b) {
        return pair(a.first * b.first % Mod.first, a.second * b.second
            % Mod.second);
    }

    void init(int n) {
        p.resize(n + 1), p[0] = pair(1, 1);
        for (int i = 1; i <= n; i++) p[i] = p[i - 1] * base;
        return;
    }

    struct Hash {
        vector <Val> h;

        Hash(const string &s) {
            h.resize(s.size() + 1);
```

```
            for (int i = 1; i <= s.size(); i++)
                h[i] = h[i - 1] * base + pair(s[i - 1], s[i - 1]);
            return;
        }

        Val getHash(int l, int r) {
            if (l > r) return pair(0, 0);
            return h[r] - h[l - 1] * p[r - l + 1];
        }

        Val val() {
            return h.back();
        }
    };
}
```

5.2 KMP

首先对字符串首先要求一个前缀函数 $\pi[i]$。$\pi[i]$ 简单来说就是子串 $s[0\ldots i]$ 最长的相等的真前缀与真后缀的长度。

```
vector<int> prefix_function(const string &s) {
    int n = s.size();
    vector<int> pi(n);
    for (int i = 1, j; i < n; i++) {
        j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

然后就是 KMP 算法的实现有两种，两种做法效率实际上一样的

```
// pattern 在 text 中出现的位置
vector<int> kmp(const string &text, const string &pattern) {
    string cur = pattern + '#' + text;
    int n = text.size(), m = pattern.size();
    vector<int> v, lps = prefix_function(cur);
```

```
    for (int i = m + 1; i <= n + m; i++)
        if (lps[i] == m) v.push_back(i - 2 * m);
    return v;
}
```

除了这样做之外，还有一种做法是求不重复的匹配位置

```
vector<int> kmp(const string &text, const string &pattern) {
    vector<int> v, lps = prefix_function(pattern);
    for (int i = 0, j = 0; i < text.size(); i++) {
        while (j && text[i] != pattern[j]) j = lps[j - 1];
        if (text[i] == pattern[j]) j++;
        if (j == pattern.size())
            v.push_back(i - j + 1), j = 0;
    }
    return v;
}
```

5.3 Tire

```
struct Trie {
    struct node {
        vector<int> nxt;
        bool exist;
        char val;

        node(char val = '@') : nxt(26, -1), exist(false), val(val) {};
    };

    vector <node> t;

    Trie() : t(1, node()) {};

    void insert(string s) {
        int pos = 0;
        for (auto c: s) {
            int x = c - 'a';
            if (t[pos].nxt[x] == -1)
                t[pos].nxt[x] = t.size(), t.emplace_back(c);
```

```
                pos = t[pos].nxt[x];
            }
            return;
        }

        bool find(string s) {
            int pos = 0;
            for (auto c: s) {
                int x = c - 'a';
                if (t[pos].nxt[x] == -1) return false;
                pos = t[pos].nxt[x];
            }
            return t[pos].exist;
        }
};
```

5.4 最小表示法

5.4.1 循环同构

如果字符串 S 选择一个位置 i 满足

$$S[i...n] + S[1...i-1] = T$$

则称 S 与 T 循环同构

5.4.2 最小表示法

对于一对字符串 A, B，他们在原串中的起始位置分别为 i, j，且前 k 个字符均相同，即

$$S[i...i+k-1] = S[j...j+k-1]$$

若 $S[i+k] > S[j+k]$，则其实下表 $l \in [i, i+k]$ 的字符串均不可能为最优解，因为如果有 $l = i+p$ 则一定有 $j+p$ 字典序更小，所以可以直接把 i 移动到 $i+k+1$ 进行比较。

```
int minNotation(const vector<int> &s) {
    int n = s.size();
    int i = 0, j = 1;
    for (int k = 0; k < n && i < n && j < n;) {
        if (s[(i + k) % n] == s[(j + k) % n]) k++;
        else {
```

```
            if (s[(i + k) % n] > s[(j + k) % n]) i = i + k + 1;
            else j = j + k + 1;
            if (i == j) i++;
            k = 0;
        }
    }
    return min(i, j);
}
```

5.4.3 Manacher

其中 p_i 表示以位置 i 为中心有 $\left\lfloor \frac{p_i}{2} \right\rfloor$ 个回文串，且最长的长度为 $p_i - 1$

```
string init(const string &s) {
    string t = "#";
    for (const char &c: s)
        t += c, t += '#';
    return t;
}

vector<int> manacher(const string &s) {
    int n = s.size();
    vector<int> p(n);
    for (int i = 0, j = 0; i < n; i++) {
        if (j + p[j] > i) p[i] = min(p[j * 2 - i], j + p[j] - i);
        while (i >= p[i] and i + p[i] < n and s[i - p[i]] == s[i + p[i]]) p[i]++;
        if (i + p[i] > j + p[j]) j = i;
    }
    return p;
}
```

第六章　动态规划

6.1　线性 DP

6.2　背包

6.2.1　01 背包输出方案

```
// dp 时
// c[i] 是价格 , w[i] 是价值
for(int i = 0 ; i < N ; i++) {
    for(int j = V ; j >= c[i] ; j++)
        if(f[j-c[i]]+w[i] > f[j]) {
            f[j] = f[j-c[i]]+w[i];
            path[i][j] = 1;
        }
}

// 输出方案

int i = N, j = V;
while(i > 0 && j > 0) {
    if(Path[i][j] == 1) {
        cout << c[i-1] << " ";
        j -= c[i-1];
    }
    i--;
}
```

6.3 树形 DP

6.3.1 普通树形 DP

> 给一颗 n 个节点有根的树，树上标号 i 的点权值为 h_i。在树上选一些点，要求父节点和子节点不能同时选。问权值和最大是多少?

$f[i][0]$ 表示在 i 子树中选择，不选 i 的最大权值和，$f[i][1]$ 表示选 i 的最大权值和。

对于一对父子 (x,y)，如果选父节点 x，则 y 不能选，`f[x][1] += f[y][0]`。如果不选 x，则 y 随意，`f[x][0] += max(f[y][1] , f[y][0])`。注意选 x 还要加上本身，`f[x][1] += h[x]`。

```
// NC1044A
#include <bits/stdc++.h>
using namespace std;
int read() { ... }

int n, root;
vector<bool> v;
vector<int> h;
vector <vector<int>> e, f;

void dp(int x) {
    f[x][1] = h[x];
    for (auto y: e[x]) {
        dp(y);
        f[x][1] += f[y][0];
        f[x][0] += max(f[y][0], f[y][1]);
    }
}
int32_t main() {
    n = read();
    v = vector<bool>(n + 1, 0), h = vector<int>(n + 1);
    e = vector < vector < int >> (n + 1);
    f = vector < vector < int >> (n + 1, vector<int>(2));
    for (int i = 1; i <= n; i++) h[i] = read();
    for (int i = 1, x, y; i < n; i++)
        x = read(), y = read(), e[y].push_back(x), v[x] = 1;
    for (int i = 1; i <= n; i++) {
        if (v[i]) continue;
```

```
            root = i;
            break;
        }
    dp(root);
    cout << max(f[root][0], f[root][1]);
    return 0;
}
```

6.3.2 背包类树形 DP

给一颗大小为 n 树，树上每个点都有一个点权。选择节点的先决条件是是选择其父节点。最多可以选择 m 个节点，问可选的最大权值是多少。

设 $f[i][j]$ 表示 i 节点子树中选择的不超过 j 个节点的最大权值。从儿子转移过来的过程其实就是一个类似**分组背包**的过程。

```
// NC1044B
void dp(int x) {
    f[x][0] = 0;
    for (auto y: e[x]) {
        dp(y);
        for (int i = m; i >= 0; i--)// 枚举当前节点最大可用背包容积
            for (int j = i; j >= 0; j--)//枚举当子节点最大可用背包容积
                f[x][i] = max(f[x][i], f[x][i - j] + f[y][j]);
    }
    // 这里要给每种容积都加上他本身的权值。
    for (int i = m; x != 0 && i > 0; i--) f[x][i] = f[x][i-1]+val[x];
    return;
}
```

6.3.3 换根 DP

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

```
// luogu P3478
#include<bits/stdc++.h>
```

```
using namespace std;
#define int long long

int read() {...}

int n;
vector<int> d, f, son;
vector<bool> vis;
vector<vector<int>> e;

void dfs(int x) {
    vis[x] = 1;
    for (auto y: e[x]) {
        if (vis[y]) continue;
        d[y] = d[x] + 1;
        dfs(y);
        son[x] += son[y];
    }
    return;
}

void dp(int x){
    vis[x] = 1;
    for( auto y : e[x] ){
        if( vis[y] ) continue;
        f[y] = f[x] + n - 2*son[y];
        dp(y);
    }
}


int32_t main() {
    n = read();
    d = vector<int>(n + 1), f = vector<int>(n + 1);
    son = vector<int>(n + 1, 1), son[0] = 0;
    vis = vector<bool>(n + 1);
    e = vector<vector<int>>(n+1);
    for (int i = 1, u, v; i < n; i++)
```

```
        u = read(), v = read(), e[u].push_back(v), e[v].push_back(u);
    dfs(1);
    vis = vector<bool>(n + 1);
    for( int i = 1 ; i <= n ; i ++ )
        f[1] += d[i];
    dp(1);
    int val = 0 , res = 0;
    for( int i = 1 ; i <= n ; i ++ )
        if( f[i] > val ) val = f[i] , res = i;
    cout << res;
    return 0;
}
```

6.3.4 最大独立集

一条边的两个端点只能选一个，问最多选多少个点。

```
// NC51178
int32_t main() {
    int n;
    cin >> n;
    vector<int> h(n + 1);
    for (int i = 1; i <= n; i++) cin >> h[i];
    vector<vector<int>> e(n + 1);
    int root = n * (n + 1) / 2;
    for (int i = 1, x, y; i < n; i++)
        cin >> y >> x, e[x].push_back(y), root -= y;
    vector<array<int, 2>> f(n+1);
    auto dfs = [e, h, &f](auto &&self, int x) -> void {
        f[x][1] = h[x];
        for (auto y: e[x]) {
            self(self, y);
            f[x][0] += max(f[y][0], f[y][1]);
            f[x][1] += f[y][0];
        }
        return;
    };
    dfs(dfs, root);
    cout << max(f[root][1], f[root][0]);
```

```
    return 0;
}
```

6.3.5 最小点覆盖

选一个点可以把相邻的边覆盖，问最少选多少个点可以把所有的边覆盖。

```
// NC 51222
int32_t main() {
    int n;
    while (cin >> n) {
        vector<vector<int>> e(n);
        vector<array<int, 2>> f(n);
        int root = n * (n - 1) / 2;
        for (int x, y, t, i = 1; i <= n; i++)
            for( x = read() , t = read() ; t ; t -- )
                y = read(), e[x].push_back(y), root -= y;
        auto dfs = [e, &f](auto &&self, int x) -> void {
            f[x][1] = 1;
            for (auto y: e[x]) {
                self(self, y);
                f[x][1] += min(f[y][0], f[y][1]);
                f[x][0] += f[y][1];
            }
        };
        dfs(dfs, root);
        cout << min(f[root][1], f[root][0]) << "\n";
    }
    return 0;
}
```

6.3.6 最小支配集

选一个点可以把相邻的点覆盖，问最少选多少个点可以把所有的点覆盖。

```
// NC 24953
int32_t main() {
    ios::sync_with_stdio(0), cin.tie(0);
    int n;
    cin >> n;
```

```cpp
    vector<vector<int>> e(n + 1);
    for (int i = 1, x, y; i < n; i++)
        cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
    vector<array<int, 3>> f(n + 1);
    // f[x][0] x 被自己覆盖,f[x][1] x 被儿子覆盖,f[x][2] x 被父亲覆盖
    auto dfs = [&f, e](auto &&self, int x, int fa) -> void {
        f[x][0] = 1;
        f[x][1] = inf;
        f[x][2] = 0;
        int inc = inf;
        for (auto y: e[x]) {
            if (fa == y) continue;
            if (f[x][1] == inf) f[x][1] = 0;
            self(self, y, x);
            f[x][0] += min({f[y][0], f[y][1], f[y][2]});
            f[x][2] += min(f[y][0], f[y][1]);
            f[x][1] += min(f[y][0], f[y][1]), inc = min(inc, f[y][0] - f[y][1]);
        }
        f[x][1] += max(0, inc);
        return;
    };

    dfs(dfs, 1, -1);
    cout << min(f[1][0], f[1][1]);
    return 0;
}
```

6.3.7 求任意子树的直径

```cpp
int32_t main() {
    int n;
    cin >> n;
    vector<int> val(n + 1);
    vector<vector<int>> e(n + 1);
    for (int i = 1; i <= n; i++) cin >> val[i];
    for (int i = 1, x, y; i < n; i++)
        cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
```

```
    vector<int> f(n + 1 , - inf ), dis(n + 1);
    // f[x] 表示 x 的子树的直径 , dis[x] 表示 x 向下最远可以走多远
    auto dfs = [e, val, &f, &dis](auto &&self, int x, int fa) -> void
       {
        multiset<int> t;
        for (auto y: e[x]) {
            if (y == fa) continue;
            self(self, y, x);
            f[x] = max(f[x], f[y]);
            dis[x] = max(dis[x], dis[y] + val[y]);
            t.insert(dis[y] + val[y]);
            if (t.size() == 3) t.erase(t.begin());
        }
        int w = val[x];
        for (auto i: t) w += i;
        f[x] = max(f[x], w);
        return;
    };
    dfs(dfs, 1, -1);
    return 0;
}
```

6.4 状态压缩 DP

6.4.1 TSP 问题

给一个有权图，求一条代价和最小的回路，使得该回路恰好经过每个点一次

复杂度 $O(n^2 2^n)$

```
// https://www.acwing.com/problem/content/93/
// 这道题给了额外的限定，要求起点必须从 0 开始
#include <bits/stdc++.h>
using namespace std;
#define int long long
const int N = 20, M = 1 << N, inf = 1e17;
int e[N][N], f[M][N], n;

int calc(int st, int i) {
    if (f[st][i] != inf) return f[st][i];
```

```
    f[st][i] --; // 标记当前状态被访问过了，以免当前状态无解被反复访问
    int p = st - (1 << i);
    for (int j = 0; j < n; j++) {
        if ((p & (1 << j)) == 0) continue;
        f[st][i] = min(f[st][i], calc(p, j) + e[j][i]);
    }
    return f[st][i];
}

int32_t main() {
    ios::sync_with_stdio(0), cin.tie(0);
    cin >> n;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> e[i][j]; // 边权
    fill(f[0], f[0] + M * N, inf) , f[1][0] = 0;
    cout << calc( ( 1 << n ) - 1 , n - 1 );
    return 0;
}
```

6.4.2 SOS DP

SOS DP 又称高维前缀和，一般用来解决以下问题

对于所有的 i, 满足 $0 \le i < 2^n$，求解 $\sum_{j \in i} a_j$

这里可以把 i 当做一个用二进制集合，每个物品只有选或不选两种情况。

先给一种非容斥的方法求解前缀和

```
// 二维
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        a[i][j] += a[i - 1][j];
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        a[i][j] += a[i][j - 1];



```

```
// 三维
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        for(int k = 1; k <= n; k++)
            a[i][j][k] += a[i - 1][j][k];
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        for(int k = 1; k <= n; k++)
            a[i][j][k] += a[i][j - 1][k];
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n; j++)
        for(int k = 1; k <= n; k++)
            a[i][j][k] += a[i][j][k - 1];
```

其实 SOS DP 也是类似做法，一维一维的做

```
// 设维度为 n 维

// 子集
for(int j = 0; j < n; j ++)
    for(int i = 0; i < (1<<n); i ++)
        if(i & (1<<j)) f[i] += f[i ^ (1<<j)];

// 超集
for(int j = 0; j < n; j ++)
    for(int i = 0; i < (1<<n); i ++)
        if(not (i & (1<<j))) f[i] += f[i ^ (1<<j)];
```

6.5 Educational DP Contest

6.5.1 A - Frog 1

有 N 块石头，编号为 $1,2,\ldots,N$。每块 i（$1 \le i \le N$），石头 i 的高度为 h_i。
有一只青蛙，它最初在石块 1 上。它会重复下面的动作若干次以到达石块 N：

- 如果青蛙目前在石块 i 上，则跳到石块 $i+1$ 或石块 $i+2$ 上。这里需要付出 $|h_i - h_j|$ 的代价，其中 j 是要降落的石块。

求青蛙到达石块 N 之前可能产生的最小总成本。

简单的线性 dp，$f[i]$ 为到达 i 的最小的代价，所以转移方程就是:

$f[i] = \min(f[i-1] - |h_i - h_{i-1}|, f[i-2] - |h_i - h_{i-2}|)$

```cpp
#include<bits/stdc++.h>

using namespace std;

#define int long long
using vi = vector<int>;
using i32 = int32_t;
using pii = pair<int, int>;
using vii = vector<pii>;

const int inf = 1e9, INF = 1e18;
const int mod = 1e9 + 7;
const vi dx = {0, 0, 1, -1}, dy = {1, -1, 0, 0};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vi h(n + 1);
    for (int i = 1; i <= n; i++) cin >> h[i];
    vi f(n + 1);
    f[1] = 0, f[2] = abs(h[2] - h[1]);
    for (int i = 3; i <= n; i++)
        f[i] = min(f[i - 1] + abs(h[i] - h[i - 1]), f[i - 2] + abs(h[i] - h[i - 2]));
    cout << f[n] << "\n";
    return 0;
}
```

6.5.2 B - Frog 2

有 N 块石头，编号为 $1, 2, \ldots, N$。每块 i（$1 \le i \le N$），石头 i 的高度为 h_i。
有一只青蛙，它最初在石块 1 上。它会重复下面的动作若干次以到达石块 N:

- 如果青蛙目前在石块 i 上，请跳到以下其中一个位置：石块 $i+1, i+2, \ldots, i+K$。这里会产生 $|h_i - h_j|$ 的代价，其中 j 是要降落的石头。

求青蛙到达石块 N 之前可能产生的最小总成本。

与上一题不同的是，这次转移的前驱很多，但依旧可以直接转移，复杂度 $O(NK)$

```
#include<bits/stdc++.h>

using namespace std;

#define int long long
using vi = vector<int>;
using i32 = int32_t;
using pii = pair<int, int>;
using vii = vector<pii>;

const int inf = 1e9, INF = 1e18;
const int mod = 1e9 + 7;
const vi dx = {0, 0, 1, -1}, dy = {1, -1, 0, 0};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vi h(n + 1);
    for (int i = 1; i <= n; i++) cin >> h[i];
    vi f(n + 1, inf);
    f[1] = 0;
    for (int i = 2; i <= n; i++)
        for (int j = max(1ll, i - k); j < i; j++)
            f[i] = min(f[i], f[j] + abs(h[i] - h[j]));
    cout << f[n] << "\n";
    return 0;
}
```

6.5.3 C - Vacation

有 N 天。每 i $(1 \le i \le N)$ 天，第 i 天有三种活动 A, B, C，只能进行一种活动，每种活动会获得 a_i, b_i, c_i 的快乐值，相邻两天的活动不能相同，请求快乐值之和的最大值。

$f[i][j]$ 表示前 i 天，且第 i 天进行活动 j 的最大快乐值。只需要 3×3 的枚举状态和前驱进行转移即可。

```cpp
#include<bits/stdc++.h>

using namespace std;

#define int long long
using vi = vector<int>;
using i32 = int32_t;
using pii = pair<int, int>;
using vii = vector<pii>;

const int inf = 1e9, INF = 1e18;
const int mod = 1e9 + 7;
const vi dx = {0, 0, 1, -1}, dy = {1, -1, 0, 0};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vector<array<int, 3>> v(n), f(n);
    for (auto &it: v)
        for (auto &i: it) cin >> i;
    f[0] = v[0];
    for (int i = 1; i < n; i++)
        for (int x = 0; x < 3; x++) {
            for (int y = 0; y < 3; y++)
                if (x != y) f[i][x] = max(f[i][x], f[i - 1][y]);
            f[i][x] += v[i][x];
        }
    cout << ranges::max(f.back()) << "\n";
    return 0;
}
```

6.5.4 D - Knapsack 1

有 N 个项目，编号为 $1, 2, \ldots, N$。对于每个 i $(1 \le i \le N)$，项目 i 的权重为 w_i，值为 v_i。

太郎决定从 N 件物品中选择一些装进背包里带回家。背包的容量为 W，这意味着所取物品的权重之和最多为 W。

求太郎带回家的物品价值的最大可能和。

01 背包

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using vi = vector<i64>;


int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vi f(m + 1);
    for (int w, v; n; n--) {
        cin >> w >> v;
        for (int i = m; i >= w; i--) f[i] = max(f[i], f[i - w] + v);
    }
    cout << f.back() << "\n";
    return 0;
}
```

6.5.5 E - Knapsack 2

有 N 个项目，编号为 $1, 2, \ldots, N$。对于每个 i $(1 \le i \le N)$，项目 i 的权重为 w_i，值为 v_i。

太郎决定从 N 件物品中选择一些装进背包里带回家。背包的容量为 W，这意味着所取物品的权重之和最多为 W。

求太郎带回家的物品价值的最大可能和。

还是 01 背包，但是本题中 W 范围非常大，无法枚举。这也用到了一个常用的优化思路，考虑 $N \times v_i \leq 10^5$，所以可以背包求出价值为 i 的最小代价，然后找到合法的最大值即可。

```cpp
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using vi = vector<i64>;

const i64 inf = 1e18;


int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    int N = n * 1e3;
    vi f(N + 1, inf);
    f[0] = 0;
    for (int w, v; n; n--) {
        cin >> w >> v;
        for (int i = N; i >= v; i--) f[i] = min(f[i], f[i - v] + w);
    }
    for (int i = N; i >= 0; i--)
        if (f[i] <= m) {
            cout << i << "\n";
            return 0;
        }
    return 0;
}
```

6.5.6 F - LCS

> 给你字符串 s 和 t。请找出一个最长的字符串，它同时是 s 和 t 的子串。

典题求 LCS 并还原。

```cpp
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<i64>;
using pii = pair<int, int>;
const i64 inf = 1e18;


i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    string a, b;
    cin >> a >> b;
    int n = a.size(), m = b.size();
    vector f(n + 1, vi(m + 1)), is(n + 1, vi(m + 1));
    vector lst(n + 1, vector<pii>(m + 1));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if (f[i - 1][j] > f[i][j - 1]) f[i][j] = f[i - 1][j], lst[
               i][j] = pair(i - 1, j);
            else f[i][j] = f[i][j - 1], lst[i][j] = pair(i, j - 1);
            if (a[i - 1] == b[j - 1] and f[i - 1][j - 1] + 1 > f[i][j
               ])
                is[i][j] = 1, f[i][j] = f[i - 1][j - 1] + 1, lst[i][j]
                    = pair(i - 1, j - 1);
        }
    string res = "";
    for (int i = n, j = m; i and j;) {
        if (is[i][j]) res += a[i - 1];
        tie(i, j) = lst[i][j];
    }
    reverse(res.begin(), res.end());
    cout << res << "\n";
```

```
    return 0;
}
```

6.5.7 G - Longest Path

> 有一个有向图 G，它有 N 个顶点和 M 条边。顶点编号为 $1, 2, \ldots, N$，对于每个 i $(1 \leq i \leq M)$，i 条有向边从顶点 x_i 到 y_i。G 不包含有向循环。
>
> 求 G 中最长有向路径的长度。这里，有向路径的长度就是其中边的数量。

因为不存在有向环，所以最长的路径起点一定入度为 0，终点一定出度为 0。想到这个结论后，比较容易想的就是在拓扑序上线性递推即可。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<i64>;
using pii = pair<int, int>;
const i64 inf = 1e18;


i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<vi> e(n + 1);
    vi inDeg(n + 1);
    for (int i = 1, x, y; i <= m; i++)
        cin >> x >> y, inDeg[y]++, e[x].push_back(y);
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (inDeg[i] == 0) q.push(i);
    vi dis(n + 1);
```

```
    for (int x; not q.empty();) {
        x = q.front(), q.pop();
        for (auto y: e[x]) {
            dis[y] = max(dis[y], dis[x] + 1);
            if (--inDeg[y] == 0) q.push(y);
        }
    }
    cout << ranges::max(dis) << "\n";
    return 0;
}
```

6.5.8 H - Grid 1

> 有一个网格，横向有 H 行，纵向有 W 列。让 (i,j) 表示从上往下第 i 行和从左往上第 j 列的正方形。
>
> 对于每个 i 和 j（$1 \leq i \leq H$，$1 \leq j \leq W$），方格 (i,j) 由一个字符 $a_{i,j}$ 来描述。如果 $a_{i,j}$ 是 ‘.，则方格 (i,j) 是一个空方格；如果 $a_{i,j}$ 是 ‘，则方格 (i,j) 是一个墙方格。可以保证方格 $(1,1)$ 和 (H,W) 是空方格。
>
> 太郎会从方格 $(1,1)$ 开始，通过反复向右或向下移动到相邻的空方格，到达 (H,W)。
>
> 求太郎从 $(1,1)$ 到 (H,W) 的路径数。由于答案可能非常大，请求取 10^9+7 的模数。

简单的二维转移

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<i64>;
using pii = pair<int, int>;
const i64 inf = 1e18;
const i64 mod = 1e9 + 7;

i32 main() {
```

```cpp
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<string> g(n);
    for (auto &i: g) cin >> i;
    vector f(n, vi(m));
    f[0][0] = (g[0][0] == '.');
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++) {
            if (i == 0 and j == 0) continue;
            if (g[i][j] == '#') continue;
            if (i > 0) f[i][j] = (f[i][j] + f[i - 1][j]) % mod;
            if (j > 0) f[i][j] = (f[i][j] + f[i][j - 1]) % mod;
        }
    cout << f[n - 1][m - 1] << "\n";
    return 0;
}
```

6.5.9 I - Coins

> 设 N 是一个正奇数。
>
> 有 N 枚硬币，编号为 $1, 2, \ldots, N$。对于每个 i $(1 \le i \le N)$，当抛掷硬币 i 时，正面出现的概率为 p_i，反面出现的概率为 $1 - p_i$。
>
> 太郎抛出了所有的 N 枚硬币。求正面比反面多的概率。

简单的概率 dp，设 $f[i][j]$ 表示前 i 个硬币 j 个正面的个数，转移如下：

$$f[i][j] = f[i-1][j-1] \times p_i + f[i-1][j] \times (1 - p_i)$$

显然可以通过倒序枚举优化掉一维空间。

答案就是 $\sum(f[n][i] \times (i > n - i))$

```cpp
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;
```

```
#define int i64

using vi = vector<i64>;
using pii = pair<int, int>;
const i64 inf = 1e18;
const i64 mod = 1e9 + 7;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n;
    vector<ldb> f(n + 1);
    f[0] = 1;
    for (ldb p, t = n; t > 0; t -= 1) {
        cin >> p;
        for (int i = f.size() - 1; i >= 0; i--) {
            f[i] *= (1.0 - p);
            if (i > 0) f[i] += f[i - 1] * p;
        }
    }
    ldb res = 0;
    for (int i = 0; i <= n; i++)
        res += (i * 2 > n) * f[i];
    cout << fixed << setprecision(10) << res << "\n";
    return 0;
}
```

6.5.10 J - Sushi

有 N 道菜，编号为 $1, 2, \ldots, N$。最初，每个 i（$1 \leq i \leq N$），i 盘都有 a_i（$1 \leq a_i \leq 3$）个寿司。$(1 \leq a_i \leq 3)$ 块寿司。

太郎会重复执行以下操作，直到所有寿司都被吃掉：

掷一个骰子，骰子上显示的数字 $1, 2, \ldots, N$ 的概率相等，结果为 i。如果骰子 i 上有几块寿司，就吃掉其中一块；如果没有，就什么都不吃。

求在所有寿司都被吃掉之前进行该操作的预期次数。

可以注意到的是选择哪个盘子无所谓，答案与盘子的顺序无关，只与盘子中剩下寿司数量有关。

可以设状态为 $f[a][b][c]$ 表示当前有 a 个盘子剩 1 个，b 个盘子剩 2 两个，c 个盘子剩三个的期望操作次数。

则有 $\frac{a}{n}$ 的概率选择剩 1 个的盘子，$\frac{b}{n}$ 的概率选到剩 2 个的盘子，$\frac{c}{n}$ 的概率选到剩 3 个的盘子，$\frac{n-a-b-c}{n}$ 的概率选到剩 0 个的盘子。

所以转移方程为

$$f[a][b][c] = 1 + \frac{a}{n}f[a-1][b][c] + \frac{b}{c}f[a+1][b-1][c] + \frac{c}{n}f[a][b+1][c] + \frac{n-a-b-c}{n}f[a][b][c]$$

可以看到 $f[a][b][c]$ 出现在了方程两侧，所以可以把方程移项得到

$$f[a][b][c] = \frac{n}{a+b+c} + \frac{a}{a+b+c}f[i-1][j][k] + \frac{b}{a+b+c}f[a+1][b-1][c] + \frac{c}{a+b+c}f[a][b+1][c-1]$$

根据这个方程便可以进行转移，但枚举比较复杂，所以可以使用深搜加记忆化实现代码

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;

#define int long long

using vi = vector<int>;
using pii = pair<int, int>;

const int inf = 1e9;
const int N = 301;
const ldb eps = 1e-6;

ldb f[N][N][N];
int n;

ldb calc(int a, int b, int c) {
    if (a == 0 and b == 0 and c == 0) return 0;
    if (f[a][b][c] >= eps) return f[a][b][c];
    ldb q = a + b + c;
    f[a][b][c] = (ldb) n / q;
    if (a > 0) f[a][b][c] += (ldb) a / q * calc(a - 1, b, c);
    if (b > 0) f[a][b][c] += (ldb) b / q * calc(a + 1, b - 1, c);
```

```
    if (c > 0) f[a][b][c] += (ldb) c / q * calc(a, b + 1, c - 1);
    return f[a][b][c];
}

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n;
    int a = 0, b = 0, c = 0;
    for (int i = 1, x; i <= n; i++) {
        cin >> x;
        if (x == 1) a++;
        else if (x == 2) b++;
        else c++;
    }
    cout << fixed << setprecision(20)<< calc(a,b,c) << "\n";
    return 0;
}
```

6.5.11 K - Stones

有一个由 N 个正整数组成的集合 $A = \{a_1, a_2, \ldots, a_N\}$。太郎和二郎将进行下面的对弈。
最初，我们有一堆由 K 个石子组成的棋子。从太郎开始，两位棋手交替进行以下操作：
在 A 中选择一个元素 x，然后从棋子堆中移走正好 x 个棋子。
当棋手无法下棋时，他就输了。假设两位棋手都以最佳状态下棋，请确定获胜者。

如果当前没有石子，则为先手必败态。所有能够一步到达先手必败的状态均为先手必胜态，无论如何都到达不了先手必败态的状态就是先手必败态。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;

#define int long long

```

```cpp
using vi = vector<int>;
using pii = pair<int, int>;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vi a(n);
    for (auto &i: a) cin >> i;
    vector<bool> f(k + 1);
    for (int i = 1; i <= k; i++)
        for (auto x: a) {
            if (i - x < 0) continue;
            if (f[i - x] == 0) {
                f[i] = 1;
                break;
            }
        }
    if (f.back()) cout << "First\n";
    else cout << "Second\n";
    return 0;
}
```

6.5.12 L - Deque

> 太郎和二郎将进行以下对弈。
>
> 最初，他们得到一个序列 $a = (a_1, a_2, \ldots, a_N)$。在 a 变为空之前，两位棋手从太郎开始交替执行以下操作：
>
> 移除 a 开头或结尾的元素。棋手获得 x 分，其中 x 为移除的元素。
>
> 假设 X 和 Y 分别是太郎和二郎在游戏结束时的总得分。太郎试图最大化 $X - Y$，而二郎试图最小化 $X - Y$。
>
> 假设两位棋手的下法都是最优的，请找出 $X - Y$ 的结果值。

设 $f[l][r][1]$ 表示区间 $[l, r]$ 中 $X - Y$ 的最大值且最后一次操作是太郎，$f[l][r][0]$ 表示区间 $[l, r]$ 中 $Y - X$ 的最大值二郎，所以转移如下：

$f[l][r][1] = \max(a[l] - f[l+1][r][0], a[r] - f[l][r-1][0])$

$\text{f[l][r][0]} = \max(a[r] - f[l+1][r][0], a[r] - f[l][r-1][0])$

然后发现两个方程的转移是完全一样的，且最终得到的结果也是一样的，所以可以省略掉第三位。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;

#define int long long

using vi = vector<int>;
using pii = pair<int, int>;

const int inf = 1e18;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vi a(n);
    for (auto &i: a) cin >> i;
    vector f(n, vi(n, -inf));
    auto calc = [&](auto &&self, int l, int r) -> i64 {
        if (l > r) return 0ll;
        if (f[l][r] != -inf) return f[l][r];
        f[l][r] = max(a[l] - self(self, l + 1, r), a[r] - self(self, l, r - 1));
        return f[l][r];
    };
    cout << calc(calc, 0, n - 1) << "\n";
    return 0;
}
```

6.5.13 M - Candies

有 N 个孩子，编号为 $1, 2, \ldots, N$。

他们决定分享 K 颗糖果。在这里，每个 i（$1 \leq i \leq N$），i 个孩子必须分到 0 到 a_i 颗糖果（包括 0 和 a_i）。另外，糖果不能剩下。

求他们分享糖果的方法数，模数为 $10^9 + 7$。在这里，如果有一个孩子得到的糖果数量不同，那么这两种方法就是不同的。

前缀和优化。

首先 $dp[i][j]$ 表示前 i 个人共分得 j 个糖果的方案数。下一个套路就是枚举第 i 个人分到的多少糖果，但是这样做复杂度太高了，转移如下

$$dp[i][j] = \sum_{k=\max(j-a[i],0)}^{j} dp[i-1][k]$$

如果我们维护出了 $dp[i-1][k]$ 的前缀和，这就可以省掉一维的枚举。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;

#define int long long

using vi = vector<int>;
using pii = pair<int, int>;
const int mod = 1e9 + 7;
const int inf = 1e18;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    vi a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];
    vector f(n + 1, vi(k + 1)), sum(n + 1, vi(k + 1));
    f[1][0] = sum[1][0] = 1;
```

```
    for (int i = 1; i <= k; i++)
        f[1][i] = i <= a[1], sum[1][i] = sum[1][i - 1] + f[1][i];
    for (int i = 2; i <= n; i++) {
        f[i][0] = sum[i][0] = 1;
        for (int j = 1; j <= k; j++) {
            if (j <= a[i]) f[i][j] = sum[i - 1][j];
            else f[i][j] = (sum[i - 1][j] - sum[i - 1][j - a[i] - 1] +
                mod) % mod;
            sum[i][j] = (sum[i][j - 1] + f[i][j]) % mod;
        }
    }
    cout << f[n][k] << "\n";
    return 0;
}
```

6.5.14 N - Slimes

有 N 个黏液排成一排。最初，左边的 i 个黏液的大小是 a_i 。

太郎正试图将所有的黏液组合成一个更大的黏液。他会反复执行下面的操作，直到只有一个粘液为止：

选择两个相邻的粘液，将它们组合成一个新的粘液。新黏液的大小为 $x+y$ ，其中 x 和 y 是合并前黏液的大小。这里需要花费 $x+y$ 。在组合粘泥时，粘泥的位置关系不会改变。

求可能产生的最小总成本。

区间 dp 模板。

$f[l][r]$ 表示区间把 $[l,r]$ 合并的最小代价。

我们枚举出 $[l,r]$ 然后枚举出分界点 mid，然后可以转移

$$f[l][r] = f[l][mir] + f[mid+1][r] + \sum a_i$$

所以我们枚举区间的时候必须要从小到大。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;
```

```

#define int i64

using vi = vector<int>;
using pii = pair<int, int>;

const int inf = 1e18;
const int mod = 1e9 + 7;
const vi dx = {0, 0, 1, -1}, dy = {1, -1, 0, 0};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vi a(n + 1);
    for (int i = 1; i <= n; i++)
        cin >> a[i], a[i] += a[i - 1];
    vector f(n + 1, vi(n + 1, inf));
    for (int i = 1; i <= n; i++) f[i][i] = 0;
    for (int len = 2; len <= n; len++)
        for (int l = 1, r = len; r <= n; l++, r++)
            for (int mid = l; mid + 1 <= r; mid++)
                f[l][r] = min(f[l][r], f[l][mid] + f[mid + 1][r] + a[r] - a[l - 1]);
    cout << f[1][n] << "\n";
    return 0;
}
```

6.5.15 O - Matching

有 N 名男性和 N 名女性，编号均为 $1, 2, \ldots, N$ 。

对于每个 i, j ($1 \le i, j \le N$)，男人 i 和女人 j 的相容性都是一个整数 $a_{i,j}$ 。如果是 $a_{i,j} = 1$ ，则男人 i 和女人 j 相容；如果是 $a_{i,j} = 0$ ，则不相容。

太郎试图做出 N 对，每对都由一个相容的男人和一个相容的女人组成。在这里，每个男人和每个女人必须正好属于一对。

求太郎能凑成 N 对的方式数，模为 $10^9 + 7$ 。

简单的状压 dp，设状态为 $f[i][t]$ 表示前 i 个男生，匹配女生状态 t 的方案数，其中 t 是一个二进制数每一位 01 表示一位女生是否完成匹配。每次只要枚举状态，然后再枚举当前男生和哪一位女生匹配即可计算出前驱状态。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = int64_t;


using vi = vector<i64>;
using pii = pair<i64, i64>;

const i64 mod = 1e9 + 7;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vector a(n, vi(n));
    for (auto &ai: a)
        for (auto &aij: ai) cin >> aij;
    int N = 1 << n;
    vector f(n+1, vi(N));
    f[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int t = 0; t < N; t++) {
            if (i != __builtin_popcount(t)) continue;
            for (int j = 0; j < n; j++) {
                if ((1 << j) & t == 0 or a[i - 1][j] == 0) continue;
                f[i][t] = (f[i][t] + f[i - 1][t ^ (1 << j)]) % mod;
            }
        }
    }
    cout << f[n][N - 1] << "\n";
    return 0;
}
```

6.5.16 P - Independent Set

有一棵树，树上有 N 个顶点，编号为 $1,2,\ldots,N$ 。对于每个 i ($1 \leq i \leq N-1$)，i -th 边连接顶点 x_i 和 y_i 。

太郎决定将每个顶点涂成白色或黑色。这里不允许将相邻的两个顶点都涂成黑色。

求 10^9+7 模中可以涂抹顶点的方法的个数。

树形 dp，$f[i][0/1]$ 表示 i 好的白或黑的方案数，然后我们可以枚举子节点，要知道白色的子节点黑白任意，黑色的子节点只有黑色。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = int64_t;

#define int i64

using vi = vector<i64>;
using pii = pair<i64, i64>;

const i64 mod = 1e9 + 7;

vector<vi> e;
vector<array<int, 2>> f;

void dfs(int x, int fa) {
    f[x][0] = f[x][1] = 1;
    for (auto y: e[x]) {
        if (y == fa) continue;
        dfs(y, x);
        f[x][1] = f[x][1] * f[y][0] % mod;
        f[x][0] = f[x][0] * (f[y][0] + f[y][1]) % mod;
    }
    return;
}

i32 main() {
```

```
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    e.resize(n + 1), f.resize(n + 1);
    for (int i = 1, x, y; i < n; i++) {
        cin >> x >> y;
        e[x].push_back(y);
        e[y].push_back(x);
    }
    dfs(1, -1);
    cout << (f[1][0] + f[1][1]) % mod;
    return 0;
}
```

6.5.17 Q - Flowers

> 有 N 朵花排成一排。对于每一朵 i （$1 \le i \le N$ ），从左边起第 i 朵花的高和美分别是 h_i 和 a_i 。这里，$h_1, h_2, \ldots, h_N$ 都是不同的。
>
> 太郎正在拔掉一些花朵，以便满足以下条件：
>
> 剩余花朵的高度从左到右单调递增。
>
> 求剩余花朵的美之和的最大值。

发现高度的值域其实很小，所以可以设状态为 $f[i][j]$ 表示前 i 朵花，且最大高度不超过 j 的美之和最大值。则有转移如下

$$f[i][j] = \max_{k=0}^{h[i]}(f[i-1][k]) + a[i]$$

然后很容易就可以压缩掉一维，然后转移就变成求前缀最大值。求前缀最大值可以用树状数组实现。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = int64_t;

#define int i64

```

```
using vi = vector<i64>;
using pii = pair<i64, i64>;

const i64 mod = 1e9 + 7;
const i64 inf = 1e18;

struct BinaryIndexedTree {
#define lowbit(x) ( x & -x )
    int n;
    vector<int> b;

    BinaryIndexedTree(int n) : n(n), b(n + 1, 0) {};

    void update(int i, int y) {
        for (; i <= n; i += lowbit(i)) b[i] = max(b[i], y);
        return;
    }

    int calc(int i) {
        int ans = 0;
        for (; i; i -= lowbit(i)) ans = max(ans, b[i]);
        return ans;
    }
};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vi h(n), a(n);
    for (int &i: h) cin >> i;
    for (int &i: a) cin >> i;
    int ans = 0;
    BinaryIndexedTree bit(n);
    for (int i = 0, tmp; i < n; i++) {
        tmp = bit.calc(h[i] - 1) + a[i];
        ans = max(ans, tmp);
        bit.update(h[i], tmp);
```

```
    }
    cout << ans << "\n";
    return 0;
}
```

6.5.18 R - Walk

有一个简单的有向图 G ，其顶点为 N ，编号为 $1, 2, \ldots, N$ 。

对于每个 i 和 j （$1 \le i, j \le N$ ），你都会得到一个整数 $a_{i,j}$ ，表示顶点 i 到 j 之间是否有一条有向边。如果是 $a_{i,j} = 1$ ，则存在一条从顶点 i 到 j 的有向边，如果是 $a_{i,j} = 0$ ，则没有。

求在 G 中长度为 K 的不同有向路径的数目，模数为 $10^9 + 7$ 。我们还将计算多次穿越同一条边的路径。

我们设 $f_i[x][y]$ 表示长度为 i 且从 x 到 y 的路径的方案数，则输入的矩阵就是 f_1。然后我们根据传递闭包得到转移如下

$$f_i[x][y] = \sum_{k=1}^{n} f_{i-1}[x][k] \times f_1[k][y]$$

很容易发现这个转移过程就是矩阵乘法

$$f_i = f_{i-1} \times f_1$$

所以可以用矩阵快速幂来解决这个问题。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = int64_t;

#define int i64

using vi = vector<i64>;
using pii = pair<i64, i64>;

const i64 mod = 1e9 + 7;
const i64 inf = 1e18;

struct matrix {
```

```
    static constexpr int mod = 1e9 + 7;
    int x, y;
    vector<vector<int>> v;

    matrix() {}

    matrix(int x, int y) : x(x), y(y) {
        v = vector<vector<int>>(x + 1, vector<int>(y + 1, 0));
    }

    void I() {// 单位化
        y = x;
        v = vector<vector<int>>(x + 1, vector<int>(x + 1, 0));
        for (int i = 1; i <= x; i++) v[i][i] = 1;
        return;
    }

    void display() { // 打印
        for (int i = 1; i <= x; i++)
            for (int j = 1; j <= y; j++)
                cout << v[i][j] << " \n"[j == y];
        return;
    }

    friend matrix operator*(const matrix &a, const matrix &b) { //乘法
        assert(a.y == b.x);
        matrix ans(a.x, b.y);
        for (int i = 1; i <= a.x; i++)
            for (int j = 1; j <= b.y; j++)
                for (int k = 1; k <= a.y; k++)
                    ans.v[i][j] = (ans.v[i][j] + a.v[i][k] * b.v[k][j]) % mod;
        return ans;
    }

    friend matrix operator^(matrix x, int y) { // 快速幂
        assert(x.x == x.y);
        matrix ans(x.x, x.y);
```

```
        ans.I();//注意一定要先单位化
        while (y) {
            if (y & 1) ans = ans * x;
            x = x * x, y >>= 1;
        }
        return ans;
    }
};

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, k;
    cin >> n >> k;
    matrix f(n, n);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            cin >> f.v[i][j];
    f = f ^ k;
    int res = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            res = (res + f.v[i][j]) % mod;
    cout << res << "\n";
    return 0;
}
```

6.5.19 S - Digit Sum

> 求在 1 和 K （含）之间满足以下条件的整数个数，模为 10^9+7：
> 十进制数位之和是 D 的倍数。

$f[pos][x]$ 表示前 pos 位和模 d 为 x 的数的个数。

然后我们设 $dp(pos, x, flag)$，其中 $flag$ 表示前 pos 是否完全与原数相同。如果相同则当前位的取值是 $[0, a[pos]]$ 否则是 $[0, 9]$。然后枚举当前位更新状态即可。

注意如果前 pos 位不与原数完全相同是，要用记忆化。

```
#include <bits/stdc++.h>

```

```
using namespace std;

using i32 = int32_t;
using i64 = int64_t;
using vi = vector<i64>;

#define int i64

const int mod = 1e9 + 7;
int d;
vi a(1);
vector<vi> f;

int dp(int pos, int x, bool flag) {
    if (pos == 0) return x % d == 0;
    if (flag and f[pos][x] != -1) return f[pos][x];
    int ans = 0, n = flag ? 9 : a[pos];
    for (int i = 0; i <= n; i++)
        ans = (ans + dp(pos - 1, (x+ i) % d, flag or i < n)) % mod;
    if (flag) f[pos][x] = ans;
    return ans;
}

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    string k;
    cin >> k >> d;
    reverse(k.begin(), k.end());
    for (auto i: k) a.push_back(i - '0');
    f = vector(a.size(), vi(d, -1));
    cout << (dp(a.size() - 1, 0, false) + mod - 1) % mod;
    return 0;
}
```

6.5.20 T - Permutation

> 设 N 是一个正整数。给你一个长度为 $N-1$ 的字符串 s ，由 $<$ 和 $>$ 组成。
>
> 求满足以下条件的 $(1, 2, \ldots, N)$ 的 $(p_1, p_2, \ldots, p_N)$ 排列的个数，模数为 10^9+7 ：
>
> 对于每个 i ($1 \leq i \leq N-1$)，如果 s 中的 i -th 字符是 $<$，则为 p_ip_{i+1} ；如果 s 中的 i -th 字符是 $>$，则为 p_ip_{i+1} 。

设状态为 $f[i][j]$ 表示 $[1,i]$ 的排列，且最后一位是 j 的方案数。

如果符号是 $<$，则 $f[i][j]=\sum_{t=1}^{j-1} f[i-1][t]$

如果符号是 $>$，则 $f[i][j]=\sum_{t=j}^{i} f[i-1][t]$

这样如果维护了一个前缀和就可以 $O(1)$ 的进行转移了。

```
#include <bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = int64_t;
using vi = vector<i64>;

#define int i64

const int mod = 1e9 + 7;


i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    string s;
    cin >> n >> s;
    s = "  " + s;
    vi sum(n + 1, 1), f(n + 1);
    sum[0] = 0;
    for (int i = 2; i <= n; i++) {
        fill(f.begin(), f.end(), 0);
        for (int j = 1; j <= i; j++) {
            if (s[i] == '<') f[j] = sum[j - 1];
            else f[j] = (sum[i - 1] - sum[j-1] + mod) % mod;
        }
        for (int j = 1; j <= n; j++)
```

```
            sum[j] = (sum[j - 1] + f[j]) % mod;
    }
    int res = 0;
    for (int i = 1; i <= n; i++)
        res = (res + f[i]) % mod;
    cout << res << "\n";
    return 0;
}
```

6.5.21 U - Grouping

> 有 N 只兔子，编号为 $1, 2, \ldots, N$ 。
>
> 对于每只 i, j ($1 \le i, j \le N$)，兔子 i 和 j 的兼容性用整数 $a_{i,j}$ 来描述。这里，$a_{i,i} = 0$ 表示每个 i （$1 \le i \le N$ ），$a_{i,j} = a_{j,i}$ 表示每个 i 和 j （$1 \le i, j \le N$ ）。
>
> 太郎要把 N 只兔子分成若干组。在这里，每只兔子必须正好属于一组。分组后，对于每只 i 和 j （$1 \le ij \le N$ ），如果兔子 i 和 j 属于同一组，太郎就能获得 $a_{i,j}$ 分。
>
> 求太郎可能得到的最高总分。

设 $f[s]$ 表示当前选择选择的兔子集合为 s 的最高总分，获得总分只有两种情况，一种是 s 所有的兔子在一组中。还有一种是 s 由至少两个组拼起来的。

对于只有一组的情况，直接统计一下就好，对于多个组拼起来的则使用枚举的子集的方式来求。

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<int>;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
```

```
    vector a(n, vi(n));
    for (auto &ai: a)
        for (auto &aij: ai)
            cin >> aij;

    const int N = 1 << n;
    vi f(N);
    for (int s = 1; s < N; s++) {
        for (int i = 0; i < n; i++) {
            if ((s & 1 << i) == 0) continue;
            for (int j = 0; j < i; j++) {
                if ((s & 1 << j) == 0) continue;
                f[s] += a[i][j];
            }
        }
        for (int i = s; i; i = (i - 1) & s)
            f[s] = max(f[s], f[i] + f[s ^ i]);
    }
    cout << f.back() << "\n";
    return 0;
}
```

6.5.22 V - Subtree

> 给一棵树，对每一个节点染成黑色或白色。
>
> 对于每一个节点，求强制把这个节点染成黑色的情况下，所有的黑色节点组成一个联通块的染色方案数，答案对 M 取模。

换根 dp。

首先可以任意选择一个点做根节点，我代码里面选择的是 1，选定根节点后，树就变成了一颗有根树。

现在对于每个点 x，如果知道了子树中的合法方案数 $f1[x]$ 和非子树节点的合法方案数 $f2[x]$，则答案就是 $f1[x] \times f2[x]$

考虑求 $f1[x]$，我们可以枚举 x 的子节点 y，则有如下转移

$$f1[x] = \Pi(f1[y] + 1)$$

然后考虑如何求 $f2[x]$，我们已知了 x 的父亲节点 fa 和兄弟节点 y，则有如下转移

$$f2[x] = f2[fa] \times \Pi(f1[y] + 1) + 1$$

其中如果要求解 $\Pi(f1[y]+1)$ 的话复杂度是 $O(N^2)$ 的，这里因为要取模，所以采用了前缀积和后缀积的方法，$pre[y]$ 表示 y 前面兄弟结点的前缀积，$suf[y]$ 表示 y 后面兄弟节点的后缀积，则 $f2[x]=f2[fa]\times pre[x]\times suf[x]+1$

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<int>;

int n, mod;
vector<vi> e;
vi f1, f2;

void dp1(int x, int fa) {
    f1[x] = 1;
    for (auto y: e[x]) {
        if (y == fa) continue;
        dp1(y, x);
        f1[x] = (f1[x] * (f1[y] + 1)) % mod;
    }
    return;
}

void dp2(int x, int fa) {
    if (e[x].empty())return;
    vi pre(e[x].size()), suf(e[x].size());
    pre.front() = suf.back() = 1;
    for (int i = 1; i < e[x].size(); i++) {
        if (e[x][i - 1] == fa) pre[i] = pre[i - 1];
        else pre[i] = (pre[i - 1] * (f1[e[x][i - 1]] + 1)) % mod;
    }
    for (int i = e[x].size() - 2; i >= 0; i--) {
        if (e[x][i + 1] == fa) suf[i] = suf[i + 1];
```

```
        else suf[i] = (suf[i + 1] * (f1[e[x][i + 1]] + 1)) % mod;
    }
    for (int i = 0; i < e[x].size(); i++) {
        if (e[x][i] == fa) continue;
        f2[e[x][i]] = (f2[x] * pre[i] % mod * suf[i] % mod + 1) % mod;
        dp2(e[x][i], x);
    }
    return;
}


i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n >> mod;
    e.resize(n + 1), f1.resize(n + 1), f2.resize(n + 1);
    for (int x, y, i = 1; i < n; i++)
        cin >> x >> y, e[x].push_back(y), e[y].push_back(x);
    dp1(1, -1);
    f2[1] = 1, dp2(1, -1);
    for (int i = 1; i <= n; i++)
        cout << f1[i] * f2[i] % mod << "\n";
    return 0;
}
```

6.5.23 W - Intervals

给定 m 条规则形如 (l_i, r_i, a_i)，对于一个 01 串，其分数的定义是：对于第 i 条规则，若该串在 $[l_i, r_i]$ 中至少有一个 1，则该串的分数增加 a_i。

你需要求出长度为 n 的 01 串中的最大分数。

$1 \le n, m \le 2 \times 10^5$，$|a_i| \le 10^9$。

线段树优化 dp。

设 $f[i][j]$ 表示前 i 个位置，且最后一个 1 的位置为 j 的最大分数。并且我们强制规定，对于 (l_k, r_k, a_k) 我们只考虑 $r_k \le i$ 的贡献。显然这里有一个合法条件一定是 $j \le i$，所以有如下的转移

$$f[i][j] = \begin{cases} \max_{1\le l<i}(f[i-1][l]) + \sum_{l_k \le j r_k = i} a_k & j = i \\ f[i-1][j] + \sum_{l_k \le j r_k = i} a_k & j < i \end{cases}$$

为什么有这样的转移呢？首先如果 $j < i$ 则前 $i-1$ 位的最后一个也一定是 j，如果 $j = i$ 则前面的最后一位是可以任意取的。

然后我们发现对于第一维只和上一位有关，所以可以优化掉一维空间。

然后我们发现转移可以分为两部分。

第一部分是 $f[i] = \max(f[j])$。

第二部分是对于所有满足 $r_k = i$ 的 (l_k, r_k, a_k)，都有 $f[j] = f[j] + a_k, (l_k \le j \le r_k)$。

对于这两部分操作，实际上就是区间最值查询和区间修改，可以使用线段树来实现。

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<int>;

struct Node {
    int l, r, val, tag;
    Node *left, *right;

    Node(int l, int r, int val, int tag, Node *left, Node *right)
            : l(l), r(r), val(val), tag(tag), left(left), right(right)
                {};
} *root;

Node *build(int l, int r) {
    if (l == r) return new Node(l, r, 0, 0, nullptr, nullptr);
    int mid = (l + r) / 2;
    Node *left = build(l, mid), *right = build(mid + 1, r);
    return new Node(l, r, 0, 0, left, right);
}

void mark(int v, Node *cur) {
    if (cur == nullptr) return;
    cur->val += v, cur->tag += v;
    return;
}

```

```
void pushdown(Node *cur) {
    if (cur->tag == 0) return;
    mark(cur->tag, cur->left), mark(cur->tag, cur->right);
    cur->tag = 0;
    return;
}

void modify(int l, int r, int v, Node *cur) {
    if (l > cur->r or r < cur->l) return;
    if (l <= cur->l and cur->r <= r) {
        mark(v, cur);
        return;
    }
    pushdown(cur);
    int mid = (cur->l + cur->r) / 2;
    if (l <= mid) modify(l, r, v, cur->left);
    if (r > mid) modify(l, r, v, cur->right);
    cur->val = max(cur->left->val, cur->right->val);
}


i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, m;
    cin >> n >> m;
    vector<array<int, 3>> a(m);
    for (auto &[l, r, v]: a)
        cin >> l >> r >> v;
    sort(a.begin(), a.end(), [&](const auto &x, const auto &y) {
        return x[1] < y[1];
    });
    root = build(1, n);
    for (int i = 1, p = 0; i <= n; i++) {
        modify(i, i, root->val, root);
        for (; p < m and a[p][1] == i; p++)
            modify(a[p][0], i, a[p][2], root);
    }
    cout << max(0ll, root->val);
```

```
    return 0;
}
```

6.5.24 X - Tower

有 N 块，编号为 $1,2,\ldots,N$ 。对于每个 i （$1\le i\le N$ ），积木块 i 的重量为 w_i ，坚固度为 s_i ，价值为 v_i 。

太郎决定从 N 块中选择一些，按照一定的顺序垂直堆叠起来，建造一座塔。在这里，塔必须满足以下条件:

- 对于塔中的每个积木块 i ，堆叠在其上方的积木块的权重之和不大于 s_i 。

求塔中所包含的图块的最大可能权重之和。

这道题算是贪心优化 dp。

首先考虑什么样的更适合放在下面? 如果 i 比 j 更适合放在下面，则 $s_i - w_k > s_j - w_i$ 也就可以化简得到 $s_i + w_i > s_j + w_j$。这样可以进行一个排序，从小到大排序。

然后我们考虑从上往下放 $f[i]$ 表示当前累计重量为 i 的最大价值。现在如果枚举到了物品 j 则有如下转移

$$f[j+w_i] = \max(f[j+w_i], f[i]+v[i])$$

其中 $j\le s_i$，这样就可以转化成经典的 01 背包问题，记得使用倒序枚举。

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<int>;

const int N = 2e4;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n;
    cin >> n;
    vector<array<int, 3>> a(n);
```

```
    for (auto &[w, s, v]: a) cin >> w >> s >> v;
    sort(a.begin(), a.end(), [&](const auto x, const auto &y) {
        return x[0] + x[1] < y[0] + y[1];
    });
    vi f(N + 1);
    for (const auto &[w, s, v]: a)
        for (int i = s; i >= 0; i--)
            f[i + w] = max(f[i + w], f[i] + v);
    cout << ranges::max(f);
    return 0;
}
```

6.5.25 Y - Grid 2

> 给一个 $H \times W$ 的网格，每一步只能向右或向下走，给出 N 个坐标 $(r_1,c_1),(r_2,c_2),...,(r_n,c_n)$，这些坐标对应的位置不能经过，求从左上角 $(1,1)$ 走到右下角 (H,W) 的方案数，答案对 10^9+7 取模。
>
> $2 \le H,W \le 10^5, 1 \le N \le 3000$

首先从一个点到 (x_1,y_1) 到 (x_2,y_2) 的路径数有 $C(x_1+y_1-x_2-y_2, x_1-x_2)$ 中。

记从 $(1,1)$ 到 (x_i,y_i) 不经过任何障碍物的路径数 $f[i]$，则有如下转移

$$f[i] = C(x_i+y_i-2, x_i-1) - \sum f[j] \times C(x_i+y_i-x_j-y_j, x_i-x_j)$$

其中 j 是比 i 更靠近起点的点。看起来是容斥掉了一个点，让实际上因为我们记录的是不经过任何障碍物的路径，所以容斥的时候是不会重复容斥的。

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;

#define int i64

using vi = vector<int>;
using pii = pair<int, int>;

```

```
const int mod = 1e9 + 7;
const int N = 2e5;

int power(int x, int y) {
    int ans = 1;
    while (y) {
        if (y & 1) ans = ans * x % mod;
        x = x * x % mod, y /= 2;
    }
    return ans;
}

int inv(int x) {
    return power(x, mod - 2);
}

vi fact, invFact;

int C(int x, int y) {
    return fact[x] * invFact[x - y] % mod * invFact[y] % mod;
}

void init() {
    fact.resize(N + 1), invFact.resize(N + 1);
    fact[0] = 1, invFact[0] = inv(1);
    for (int i = 1; i <= N; i++)
        fact[i] = fact[i - 1] * i % mod, invFact[i] = inv(fact[i]);
    return;
}

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    init();
    int h, w, n;
    cin >> h >> w >> n;
    vector<pii> a(n);
    for (auto &[x, y]: a) cin >> x >> y;
    sort(a.begin(), a.end(), [&](const auto &x, const auto &y) {
```

```
        return x.first + x.second < y.first + y.second;
    });
    a.emplace_back(h, w);
    vi f(n + 1);
    for (int i = 0; i <= n; i++)
        f[i] = C(a[i].first + a[i].second - 2, a[i].first - 1);
    for (int i = 0; i <= n; i++) {
        auto &[xi, yi] = a[i];
        for (int j = 0; j <= n; j++) {
            auto &[xj, yj] = a[j];
            if (i == j or xi < xj or yi < yj) continue;
            f[i] = (f[i] - f[j] * C(xi + yi - xj - yj, xi - xj) % mod
                + mod) % mod;
        }
    }
    cout << f[n] << "\n";
    return 0;
}
```

6.5.26 Z - Frog 3

有 N 块石头，编号为 $1,2,...,N$ 。每块 i （$1 \leq i \leq N$ ）石头 $1 \leq i \leq N$ ），石头 i 的高度为 h_i 。这里，$h_1 h_2 \cdots h_N$ 成立。

有一只青蛙，它最初位于石块 1 上。它会重复下面的动作若干次以到达石块 N ：

- 如果青蛙目前在 i 号石块上，请跳到以下其中一块：石块 $i+1, i+2, \ldots, N$ 。这里需要花费 $(h_j - h_i)^2 + C$ ，其中 j 是要落脚的石头。

求青蛙到达石块 N 之前可能产生的最小总成本。

$2 \leq N \leq 2 \times 10^5$

斜率优化 dp。

设 $f[i]$ 表示从 1 到 i 的最小花费，这可以写出最基础的状态转移为

$$f[i] = \min(f_j + (h_i - h_j)^2 + C)$$

内部可以展开为

$$f[i] = \min(f_j h_i^2 + h_j^2 + C - 2h_i h_j)$$

把无关项提出来，可以得到

$$f[i] = h_i^2 + C + min(f_j + h_j^2 - 2h_i h_j)$$

令 $g_i = f_i + h_i^2$，则有

$$f[i] = h_i + C + min(g_j - 2h_ih_j)$$

至此，关于式子的化简就结束了，现在对于转移，如果有两个点 x, y，若满足 $x < y$ 且 x 更优，则有

$$g_x - 2h_ih_x < g_y - 2h_ih_y$$

移项得

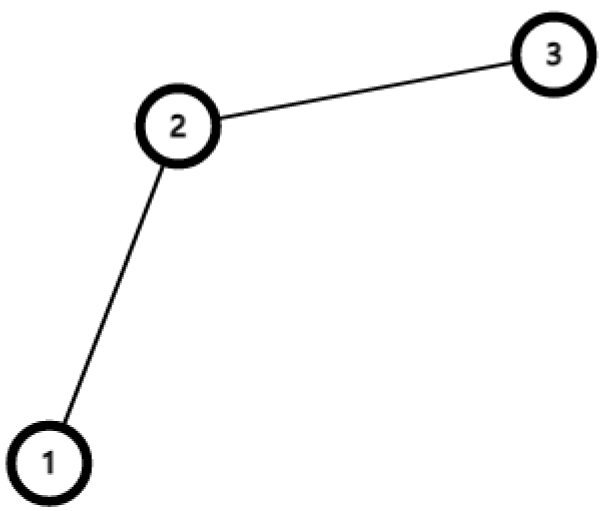
$$\frac{g_x - g_y}{h_x - h_y} > 2h_i$$

发现这个式子和斜率很像，所以令 $slop(x, y) = \frac{g_x - g_y}{h_x - h_y}$。

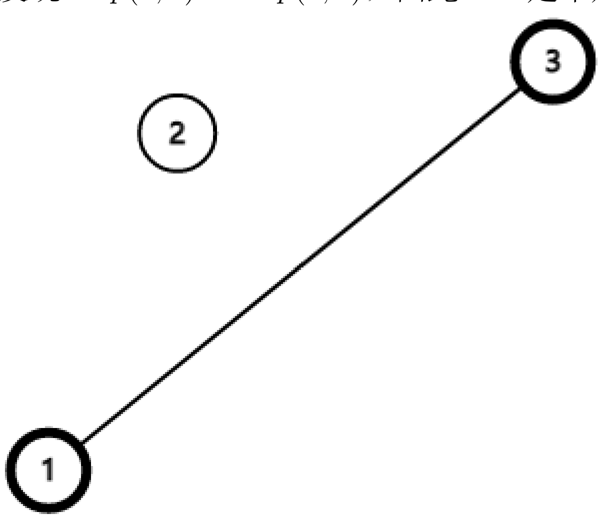
下面就开始时斜率优化的部分，有三个点 $1 < 2 < 3$，如果 2 是最优的则有

$$slop(1, 2) < 2h_i slop(2, 3) > 2h_i \Rightarrow slop(1, 2) < slop(2, 3)$$

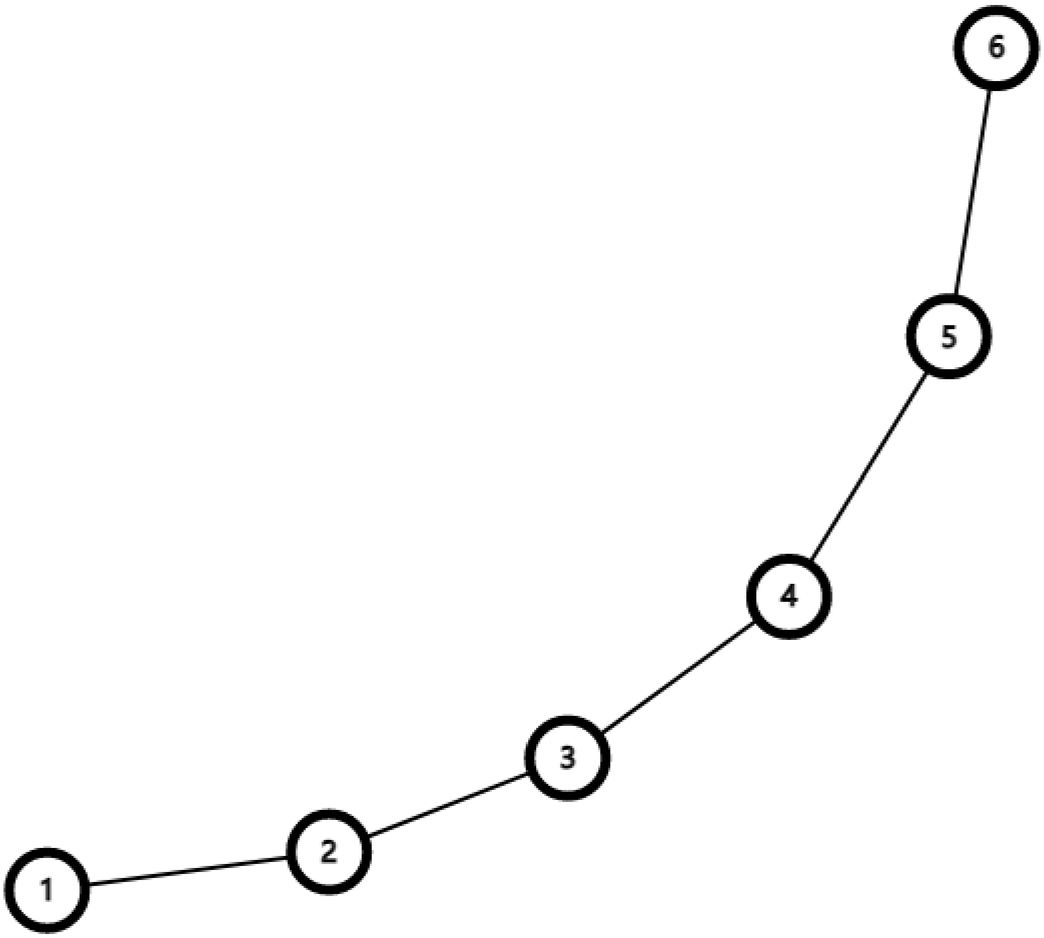
对于下图



发现 $slop(1, 2) > slop(2, 3)$，因此 2 一定不是最优的，所以可以优化成



最后你会发现，实际上就是维护一个凸包



然后，因为本题的 h_i 时保证递增的，所以最优解只会出现在队首

```
#include<bits/stdc++.h>

using namespace std;

using i32 = int32_t;
using i64 = long long;
using ldb = long double;

#define int i64

using vi = vector<int>;
using pii = pair<int, int>;

i32 main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int n, C;
    cin >> n >> C;
```

```
    vi h(n + 1), f(n + 1), g(n + 1);
    for (int i = 1; i <= n; i++) cin >> h[i];
    auto slop = [&](int x, int y) {
        return ldb(g[x] - g[y]) / ldb(h[x] - h[y]);
    };
    deque<int> q;
    q.push_back(1), g[1] = h[1] * h[1];
    for (int i = 2, j; i <= n; i++) {
        while (q.size() >= 2 and slop(q.front(), *(q.begin() + 1)) <=
           2 * h[i]) q.pop_front();
        j = q.front();
        f[i] = h[i] * h[i] + C + g[j] - 2 * h[i] * h[j];
        g[i] = f[i] + h[i] * h[i];
        while (q.size() >= 2 and slop(*(q.rbegin() + 1), q.back()) >=
           slop(q.back(), i)) q.pop_back();
        q.push_back(i);
    }
    cout << f[n];
    return 0;
}
```

战 术

> 读新题的优先级高于一切
>
> 交题之前看一遍 clarification
>
> 构造题不要开场做
>
> 开场一小时后，开始做计算几何、模拟
>
> 每道题需要至少两个人确定题意
>
> 上机前做法要得到队友的确认
>
> 题目不会但过了一片应尝试冲一发暴力
>
> 交题后立刻打印让出机器
>
> 检查清空，longlong，取模，-0.000
>
> 中后期应该一人写，一人辅助，及时发现错误
>
> 卡题了，上厕所，洗脸
>
> 卡题超过半个小时要考虑换题
>
> ACM 并不是唯一，拥抱队友，热爱生活